



AUTOMATED VIRTUAL MACHINE INTROSPECTION
FOR HOST-BASED INTRUSION DETECTION

THESIS

Brett A. Pagel, Captain, USAF

AFIT/GCE/ENG/09-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AUTOMATED VIRTUAL MACHINE INTROSPECTION
FOR HOST-BASED INTRUSION DETECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Brett A. Pagel, B.S. Computer Engineering

Captain, USAF


March 2009

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

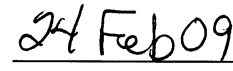
AUTOMATED VIRTUAL MACHINE INTROSPECTION
FOR HOST-BASED INTRUSION DETECTION

Brett A. Pagel, B.S. Computer Engineering
Captain, USAF

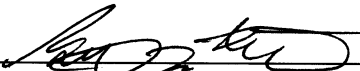
Approved:



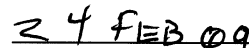
Dr. Barry E. Mullins (Chairman)



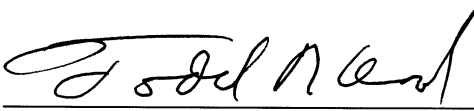
date



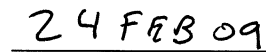
Dr. Gilbert L. Peterson (Member)



date



Maj Todd R. Andel, Ph.D. (Member)



date

Abstract

This thesis examines techniques to automate configuration of an intrusion detection system utilizing hardware-assisted virtualization. These techniques are used to detect the version of a running guest operating system, automatically configure version-specific operating system information needed by the introspection library, and to locate and monitor important operating system data structures. This research simplifies introspection library configuration and is a step toward operating system independent introspection.

An operating system detection algorithm and Windows virtual machine system service dispatch table monitor are implemented using the Xen hypervisor and a modified version of the XenAccess library. All detection and monitoring is implemented from the Xen management domain. Results of the operating system detection are used to initialize the XenAccess library. Library initialization time and kernel symbol retrieval are compared to the standard library. The algorithm is evaluated using nine versions of the Windows operating system. The system service dispatch table monitor is evaluated using the Agony and ProAgent rootkits.

The automation techniques successfully detect the operating system and system service dispatch table hooks for the nine Windows versions tested. The modified XenAccess library exhibits an average initialization speedup of 1.9. Kernel symbol lookup is 10 times faster, on average. The hook detector is able to detect all hooks used by both rootkits.

Acknowledgements

I would like to thank Bryan Payne for starting the XenAccess project. Without this library, my research would have been much more difficult. I would also like to thank my advisor, committee members, and fellow students for helping me through the research process. Last, but not least, I would like to thank my wife for her love, support, and encouragement.

Brett A. Pagel

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
Table of Contents	vi
List of Figures	ix
List of Tables	xi
List of Abbreviations	xii
 I. Introduction	 1
1.1 Motivation	1
1.2 Goals	2
1.3 Thesis Layout	2
 II. Background Research	 3
2.1 Virtualization	3
2.1.1 Paravirtualization	4
2.1.2 Binary Translation	5
2.1.3 Hardware-Assisted Virtualization	6
2.1.4 Xen Virtual Machine Monitor	8
2.2 Windows Data Structures	10
2.2.1 Microsoft Portable Executable and Common Ob- ject File Format	10
2.2.2 System Service Dispatch Table	12
2.2.3 Process and Thread Lists	12
2.3 Malware	14
2.3.1 Rootkits	15
2.3.2 Hooking	16
2.3.3 Direct Kernel Object Manipulation	19
2.4 Software Host-Based Intrusion Detection	20
2.4.1 Data Collection	20
2.4.2 Data Classification	21
2.4.3 Host-Based IDS Limitations	23
2.5 Hardware Host-Based Intrusion Detection Methods	25
2.5.1 CuPIDS	25
2.5.2 Co-Pilot	27
2.6 Virtual Machine Introspection	28

	Page
2.6.1 Explicit Information Introspection	28
2.6.2 Implicit Information Introspection	31
2.7 Summary	34
III. Methodology	35
3.1 Problem Definition	35
3.2 Goals	35
3.3 Approach	36
3.4 Experimental Design	37
3.4.1 Operating System Detection	37
3.4.2 SSDT Location and Hook Detection	42
3.5 Summary	47
IV. Results and Analysis	48
4.1 Results and Analysis of Experiments	48
4.1.1 Operating System Detection Analysis	48
4.1.2 SSDT Location and Hook Detection Analysis	53
4.2 Overall Analysis	55
4.3 Limitations	55
4.4 Summary	56
V. Conclusions	57
5.1 Research Conclusions	57
5.2 Research Impact	58
5.3 Recommendations for Future Work	58
5.3.1 Version Specific Operating System Offsets	58
5.3.2 OS Data Structure Monitoring	59
5.3.3 System Call Interposition	59
5.3.4 Linux OS Detection	59
5.4 Summary	59
Appendix A. Symbol Lookup Data	60
Appendix B. Windows Operating System Data Structures	72
B.1 _EPROCESS Structure for Windows XP SP2	72
B.2 _KPCR Structure	78
B.3 _DBGKD_GET_VERSION64 Structure	78
B.4 _KDDEBUGGER_DATA64 Structure	79

	Page
Appendix C. Building the Test Platform	82
C.1 Install Fedora 8	82
C.2 Install Xen 3.1.4	82
C.3 Configuring Xen Boot	83
C.4 XenAccess	84
C.5 Creating a HVM DomU for Xen	84
Bibliography	85

List of Figures

Figure		Page
2.1	Intel 4-Ring Protection Model	3
2.2	Virtualization Protection Model	4
2.3	Binary Translation Versus HAV Nanobenchmarks	6
2.4	VM Guest Interaction Life Cycle	7
2.5	Xen 3.0 Architecture	10
2.6	Windows XP SP2 Dump & PE File Format	11
2.7	System Service Dispatching	13
2.8	Process and Thread Lists	13
2.9	Malware Taxonomy	15
2.10	A User-Space Inline Hook Example	17
2.11	System Service Dispatch Table	18
2.12	Race Exploit	24
2.13	CuPIDS Architecture	26
2.14	Livewire Architecture	29
2.15	VMWatcher Architecture	31
2.16	Lares Architecture	32
2.17	x86 Virtual Address Translation	33
3.1	OS Detection Process	39
3.2	Shadow SSDT Location	44
4.1	Comparison of Initialization Times - 95% Confidence Interval .	50
4.2	Average CPU Ticks for Kernel Symbol Lookup	53
A.1	Symbol Lookup Comparison for Windows 2000 SP4	63
A.2	Symbol Lookup Comparison for Windows XP	64
A.3	Symbol Lookup Comparison for Windows XP SP1	65
A.4	Symbol Lookup Comparison for Windows XP SP1a	66

Figure		Page
A.5	Symbol Lookup Comparison for Windows XP SP2	67
A.6	Symbol Lookup Comparison for Windows XP SP3	68
A.7	Symbol Lookup Comparison for Windows 2003	69
A.8	Symbol Lookup Comparison for Windows 2003 SP1	70
A.9	Symbol Lookup Comparison for Windows Vista Business . . .	71

List of Tables

Table		Page
3.1	XenAccess Required Data Structure Information	37
3.2	Operating System Base Addresses	41
3.3	Agony Rootkit Options and Hooked System Calls	46
4.1	Operating System Detection Results	49
4.2	1-sample t-Test for Initialization Time Comparison	51
4.3	Hooks Detected for Agony Rootkit	54
4.4	Hooks Detected for ProAgent Rootkit	55
A.1	1-Sample t-Test of Symbol Lookup Times - Standard Library .	61
A.2	1-Sample t-Test of Symbol Lookup Times - Modified Library .	62

List of Abbreviations

Abbreviation		Page
OS	Operating System	1
HAV	Hardware-Assisted Virtualization	1
VMM	Virtual Machine Monitor	3
VM	Virtual Machine	4
AMD	Advanced Micro Devices	6
VT	Virtualization Technology	6
SVM	Secure Virtual Machine	6
VMX	Virtual Machine Extensions	6
VMCS	Virtual Machine Control Structure	7
PE	Portable Executable	10
COFF	Common Object File Format	10
DLL	Dynamic Link Library	10
SSDT	System Service Dispatch Table	12
GDI	Graphics Device Interface	12
DKOM	Direct Kernel Object Manipulation	15
API	Application Programming Interface	16
IAT	Import Address Table	16
EAT	Export Address Table	16
IDT	Interrupt Descriptor Table	18
MSR	Model Specific Register	19
IRP	Input/Output Request Packet	19
IDS	Intrusion Detection System	20
NIDS	Network Intrusion Detection Systems	20
HIDS	Host-Based Intrusion Detection	20
FSA	Finite State Automation Machine	22

Abbreviation		Page
PC	Program Counter	22
CuPIDS	Co-Processor-Based Intrusion Detection System	25
MMU	Memory Management Unit	31
TLB	Translation Lookaside Buffer	32
ASID	Address Space Identifier	32
LRU	Least Recently Used	52

AUTOMATED VIRTUAL MACHINE INTROSPECTION FOR HOST-BASED INTRUSION DETECTION

I. Introduction

1.1 *Motivation*

The privilege level structure of computer hardware and software creates a battle between malware writers and security researchers for control of the system. Programs with higher privileges, whether malicious or not, have visibility into and can control programs with lower privileges. Typically the programs that run at the highest privilege level are the operating system (OS) kernel and its components. Malware exploiting the kernel or its components can gain access to this level. Once the malware is running at the highest privilege level it can modify data and mask its presence from the kernel. The introduction of hardware-assisted virtualization technology provides yet another level for malware and security software to contend for.

Hardware-assisted virtualization (HAV) technology introduces a new highest level in the privilege hierarchy. This level is more privileged than the operating system kernel and any kernel-mode malware that might infect it. This advantage can be used to develop a secure monitor to detect malware. This new level in the privilege hierarchy also introduces its own set of challenges. The separation and isolation provided by HAV creates a “semantic gap” that makes it difficult interpret the memory of lower privilege levels [CN01].

Bridging the semantic gap requires detailed knowledge of the OS structure. For closed-source operating systems, this information can be difficult to obtain and may change with patches, updates, or new versions. One method of bridging the semantic gap, virtual machine introspection, is implemented by the XenAccess library [PCL07]. The library requires detailed OS knowledge via a configuration file and system map file. Additionally, it must scan memory to locate the base address of the kernel. This

research aims to streamline XenAccess library configuration and use it to implement secure monitoring for operating system data structures using HAV.

1.2 Goals

Configuring of a virtualization-based security monitor for several different operating systems can be difficult and tedious. Version-specific information must be obtained and cataloged for each operating system version. The goal of this research is to automatically configure the XenAccess library and use it to monitor specific OS data structures from the Xen management domain. In order to achieve this goal, the following obstacles must be overcome:

- Detect the Guest Operating System
- Identify Version-Specific Offsets
- Locate OS Data Structures to Be Monitored

The ideal solution to this goal would be a security application that can monitor any operating system and automatically monitor any important data structure. To limit the scope of the problem, this research concentrates on the Windows operating system as it has an 88.7% operating system market share [App08]. Moreover, only support for monitoring the system service dispatch table is implemented because it is exploited in approximately 50 percent of malware attacks [KS07, Rie06]. Techniques used by forensic researchers and malware writers are used to overcome these obstacles.

1.3 Thesis Layout

This chapter introduces the research motivation and goals. Chapter 2 discusses relevant research accomplished by others and background information. Chapter 3 describes the methodology for the experiments conducted for this research. Chapter 4 discusses experimental results and analyzes their outcomes. Finally, Chapter 5 provides a concluding discussion of this research and gives recommendations for future work.

II. Background Research

This chapter introduces research accomplished by others and fundamental concepts relevant to virtualization, host-based intrusion detection, and operating systems. First, Section 2.1 describes and compares several virtualization methods. Section 2.2 details important Windows operating system data structures. Section 2.3 defines malware and discusses their methods. Sections 2.4 and 2.5 present software and hardware host-based intrusion detection system technologies. Section 2.6 explores the topic of virtual machine introspection.

2.1 *Virtualization*

Virtualization provides an additional layer of abstraction beyond the Intel 4-ring protection model shown in Figure 2.1 [Cor08a]. In the Intel protection model, ring 0 is the most privileged level and is typically where the operating system kernel resides. User-level applications usually run at level three, the least privileged level. As shown in Figure 2.2, the new layer provided by virtualization resides below ring 0 and therefore at a higher privilege level. This new level interacts directly with hardware while providing an interface to ring 0 that is nearly indistinguishable from the standard hardware interface. This interface is implemented in a virtual machine monitor (VMM), or hypervisor. These can be complex programs that perform many

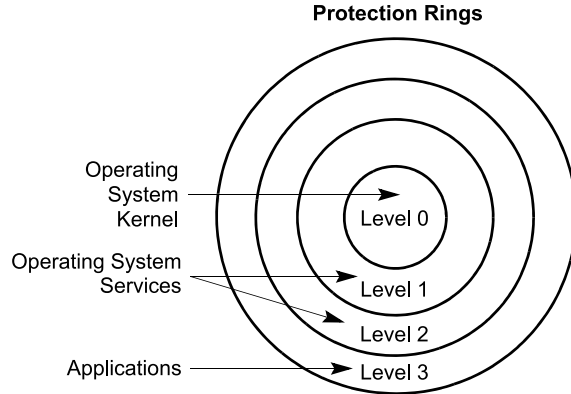


Figure 2.1: Intel 4-Ring Protection Model [Cor08a]

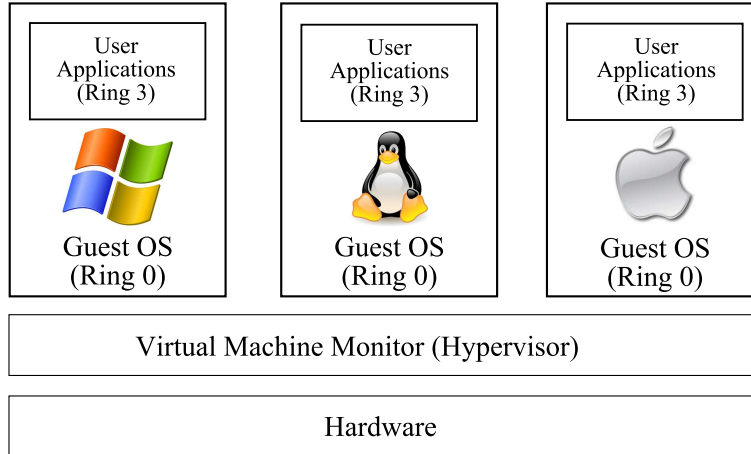


Figure 2.2: Virtualization Protection Model

functions to interact with hardware and the OS or very lightweight with minimal functionality and less chance for exploitable software bugs [Rut08].

The addition of the VMM layer increases security by isolating the operating system within a virtual machine (VM). This separation also allows for the VMM to run multiple VMs at the same time while keeping them isolated from each other. The VMM controls what each operating system is allowed to access and arbitrates any interaction between them. Server virtualization is becoming increasingly popular because of this ability to consolidate several software services onto a single hardware system. The following paragraphs, discuss the different types of virtualization and their tradeoffs.

2.1.1 Paravirtualization. Paravirtualization requires modifying operating system source code to replace privileged function calls with appropriate substitutes that result in a call to the VMM. This technique allows for the most flexibility because any potentially unsecure calls or instructions can be redirected. Moreover, additional functionality, such as VMM to OS communication, can be added as necessary. The use of paravirtualization requires modification to the guest operating system because the VMM does not reproduce the full functionality of the underlying hardware [BDF⁺03]. The major benefit of a paravirtualization is that because the OS must be modified, the

system can be designed to increase performance and to bridge the “semantic gap” by providing explicit information to the VMM which is identical to that within the OS. Jones calls this “the gold standard of OS information within a VMM,” but cautions that this is not a secure means of obtaining OS information because a compromised OS can report false information [JADAD06].

2.1.2 Binary Translation. Binary translation is used by commercial virtualization products from VMWare. Unlike paravirtualization, the operating system code does not have to be modified to cooperate with the hypervisor. Virtual CPU state is maintained separately from physical CPU state. When translating the binary instructions, the translator can intercept privileged instructions and replace them with the appropriate instruction referencing virtual data structures, but allows non-privileged instructions to execute unmodified. In addition to privileged instructions, Adams and Agesen list three other instruction types that need to be translated to maintain control of the guest operating system. These include program counter relative addressing, direct control flow, and indirect control flow [AA06]. These instruction types must have their branch targets translated to the correct address.

The translation process is efficient because the majority of code does not need to be modified and can be sent to the processor as is. The amount of overhead is dependent on how many instructions need to be translated to a different instruction(s). Adams and Agesen measure the slowdown caused by a software VMM using binary translation and a hardware-assisted virtualization VMM relative to native execution. When executing the SPECint 2000 benchmark, the software VMM and hardware-assisted VMMs have an average slowdown of 4% and 5% respectively [AA06]. SPECint 2000 is largely user-level code and therefore only incurs a small overhead. Adams and Agesen also developed seven “virtualization nanobenchmarks” to measure overhead caused by instructions or sequences of instructions that require VMM intervention. Figure 2.3 compares native, software VMM, and hardware-assisted VMM execution time of the seven benchmarks. In Figure 2.3 the number of execution cycles for native,

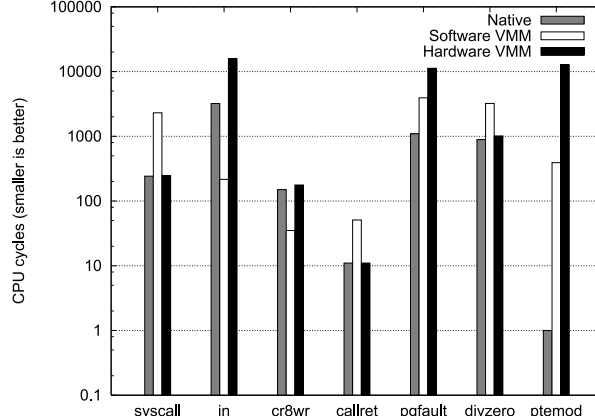


Figure 2.3: Binary Translation Versus HAV Nanobenchmarks [AA06]

software VMM, and hardware-assisted VMM execution are represented by gray, white, and black respectively. Overall, the software VMM using binary translation executes slower than native, but is faster for two of the nanobenchmarks due to a more efficient process than with native execution.

2.1.3 Hardware-Assisted Virtualization. While software designers have developed very capable suites for virtualization, many have been developed with an emphasis on performance and consolidation and do not necessarily concentrate on security. This section discusses hardware that has evolved from the lessons learned and shortfalls of software virtualization. Intel and Advanced Micro Devices (AMD) have extended their x86 architectures with hardware virtualization instructions. The hardware extensions from both manufacturers eliminate the need for the techniques used by software virtualization, such as binary translation and paravirtualization. In contrast to software virtualization where either the VMM must intercept instructions or the operating system must be modified to deprive privileged instructions, with Intel Virtualization Technology (VT) and AMD Secure Virtual Machine (SVM) the instructions are intercepted by hardware.

2.1.3.1 Intel Virtualization Technology. The Intel VT architecture is based on virtual machine extensions (VMX) that introduce ten VMX instructions to

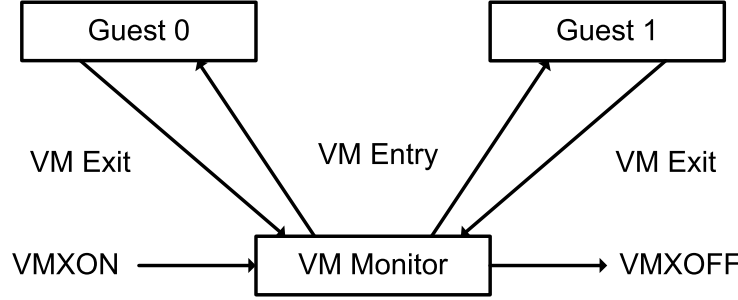


Figure 2.4: VM Guest Interaction Life Cycle [Cor08a]

the Intel 32-bit instruction set [Cor08b]. With VMX, two new CPU modes are defined- VMX root and VMX non-root. The intent of the designers is for the VMM to run at VMX root and for guest operating systems to run at VMX non-root. Additionally, in both VMX root and VMX non-root, the processor can run in any of the four privilege levels, but they are independent of each other. Ring 0 in VMX non-root is less privileged than Ring 0 in VMX root, and any privilege level changes in VMX non-root mode may cause a VM exit for servicing by the VMM.

VMX is enabled or disabled using the VMXON or VMXOFF instructions as illustrated by the VMM life cycle shown in Figure 2.4 [Cor08b]. After VMX mode is enabled, the processor transitions from root to non-root mode during virtual machine entry and exits. The VM can make service calls to the VMM via the VMCALL instruction, causing a VM exit. Execution in VMX non-root mode and VMX transitions are governed by the virtual-machine control structure (VMCS) until the VMM issues a VMXOFF instruction. The VMCS is referenced with a physical address for efficiency and contains six logical sections of control data for the VM [UNR⁺05]. The sections are guest state, host state, VM-execution control, VM-exit control, VM-entry control, and VM-exit information [Cor08b]. The VMCS can be read and written to using the VMREAD and VMWRITE instructions.

The guest state section of the VMCS contains register state and non-register state values: Activity State, Interruptibility State, Pending debug exceptions, and the VMCS link pointer. Guest state information is loaded from the guest state sec-

tion on every VM entry and saved for VM exits [Cor08b]. The host state section contains specific register values to restore the state of the processor whenever a VM exit transitions the processor back to VMX root mode [Cor08b]. The VM-execution control section details how interrupts and exceptions should be handled. In addition, it contains several bitmaps that define what should and should not cause a VM exit [UNR⁺05]. The VM-entry and VM-exit control sections of the VMCS define the basic operation of VM entry and exits. Finally, the VM-exit information section contains information about the most recent VM exit, such as the reason for exiting.

2.1.3.2 AMD Secure Virtual Machine. The AMD SVM is very similar to Intel’s VT. SVM provides a set of hardware extensions for virtualization and security. The parallel to VT’s root and non-root mode is host and guest mode in AMD’s SVM. Also, AMD’s documentation refers to VM entry and VM exit as world switching [AMD08]. The processor enters guest mode by executing a VMRUN.

SVM defines a Virtual Machine Control Block similar to VT’s VMCS that regulates how each VM functions. The VMCB is a fixed length of 2,564 bytes and contains control and state information for the associated VM [MY07]. The VMCB can be modified using the VMSAVE and VMLOAD instructions. The control section contains the *intercept vector* which tells which instructions will cause a #VMEXIT as well as rules for interrupts and exceptions. The state section of the VMCB stores register contents and other state information for the guest VM. Finally, like VT, an EXITCODE is stored telling the VMM why the guest caused an exit to host mode [AMD08].

2.1.4 Xen Virtual Machine Monitor. Xen is a virtual machine monitor that originated from the XenoServers project at the University of Cambridge [vH08]. The initial public release of Xen, version 1.0, is a paravirtualization-only VMM. Xen contains a management domain 0, or dom0, that relieves the VMM of high level VM management tasks and controls “their associated scheduling parameters, physical memory allocations and the access they are given to the machine’s physical disks and

network devices.” [BDF⁺03] Xen paravirtualized guest OSes must be modified such that all interaction with hardware is conducted through the VMM. Device drivers are replaced with Xeno-Aware drivers that interface the VMM and privileged instructions within the OS are paravirtualized with calls to the Xen hypervisor. The cost of making these changes to the guest OSes is 2,996 and 4,620 lines of code for Linux and Windows respectively [BDF⁺03].

With the release of Xen 3.0, hardware virtualization instructions are supported. This allows Xen to run unmodified operating systems reducing the overhead of paravirtualization and enabling the use of closed-source OSes. Hardware-assisted virtualization is accomplished by running the Xen VMM in VMX root mode and the guest OS in VMX non-root mode. Instructions that were paravirtualized previously will cause a VM-exit to the VMM when executed by a guest OS. Additionally, the VMM emulates devices for the guest OS so that specialized Xen device drivers are not necessary [PFH⁺05].

Figure 2.5 demonstrates the capabilities of the Xen 3.0 architecture. Domain 0 is the trusted domain handling communication between the hypervisor and untrusted VMs. It contains native device drivers with direct access to hardware. VM 1 is a trusted VM that has been given special access to the hardware via the hypervisor. VM 2 is a symmetric multiprocessor paravirtualized guest OS that has been modified to work with the Xen hypervisor. Finally, VM 3 is an unmodified Windows XP operating system using HAV. The arrows demonstrate how the domain 0 and trusted VM 1 arbitrate communication with the hardware for the other VMs.

In the 6 years since Xen 1.0 was released it has become a popular virtualization platform. It is used by thousands in both academia and the commercial sectors [vH08]. Commercial versions exist that provide virtualization solutions to corporations, but the Xen project is still an open source venture supported by programmers from over 20 well known information technology corporations [CS09]. The availability of source code makes Xen an attractive choice for research.

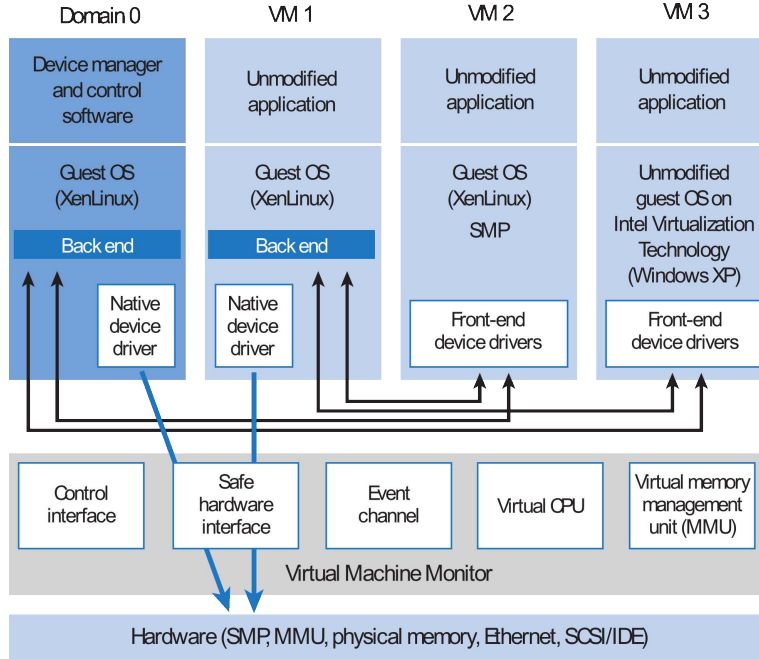


Figure 2.5: Xen 3.0 Architecture [ADC05]

2.2 Windows Data Structures

Operating systems contain many important data structures that directly control execution and are therefore targets for malicious code. In order to protect them, their function must be thoroughly understood. The following sections describe the structure and function of Windows OS data structures commonly exploited by malware.

2.2.1 Microsoft Portable Executable and Common Object File Format. The Microsoft Portable Executable (PE) and Common Object File Format (COFF) is used by executable files, dynamic link libraries (DLL), and device drivers for Windows operating systems [Cor08d]. Files that follow this specification have a defined structure that can be used to interpret them both on disk and in memory. Figure 2.6 presents the specified structure of a PE file with a memory dump of the first 656 bytes of the Windows XP Service Pack 2 kernel executable for illustration. The base of the image header begins with the hex bytes 0x4d5a, or “MZ” in ascii. Additionally, the PE header offset, highlighted in white with the value 0xe8, is always located at 0x3c and points to the location of the PE header further in the file. A valid PE image

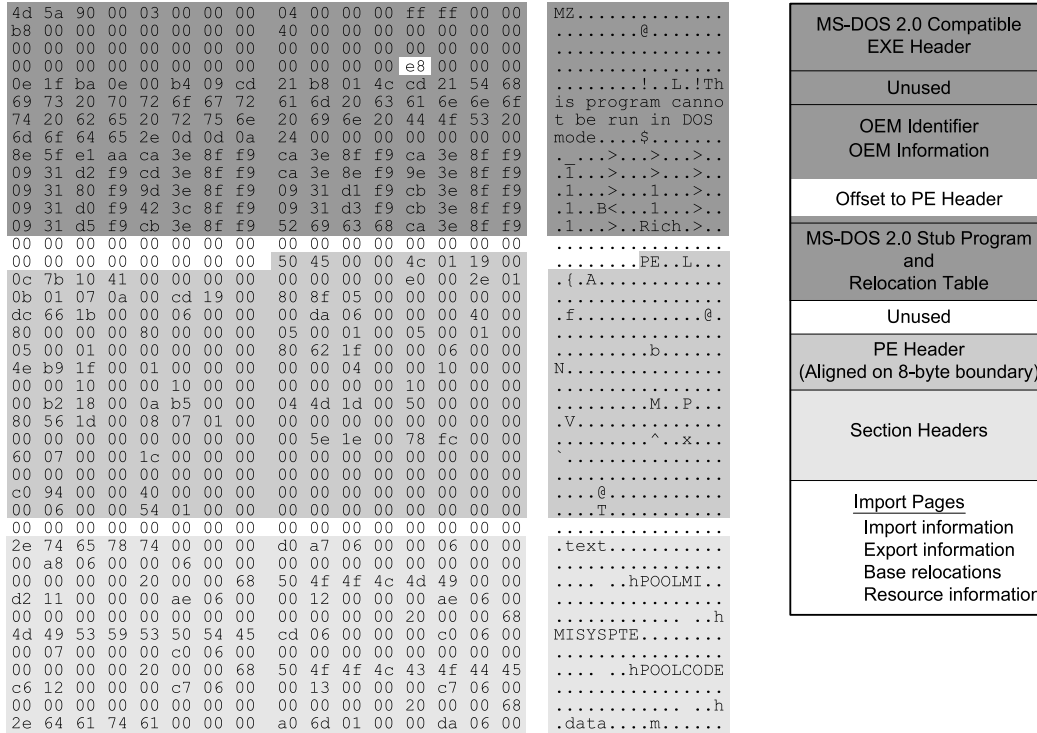


Figure 2.6: Windows XP SP2 Dump & PE File Format

begins the PE header with the 4-byte signature of “PE\0\0”, as seen in the first four bytes of the middle box. The PE header contains machine type information, section information, symbol information, size of the optional header, and file attribute flag information. Windows executables also contain an optional header appended to the PE header. The optional header begins with the “magic” hexadecimal number 0x10b for a PE32 executable, such as `ntoskrnl.exe`. The information of interest in the optional header is the data directory information. This gives the location and size of tables used by the image during execution. The resource table is of specific interest for identifying the operating system version [Car07]. Additionally, the export table contains names and pointers to all of the functions exported by the image.

Following the PE header are the section headers, in the bottom and lightest gray box, containing name, size, and location information for each section contained in the file. The resource and export table addresses are duplicated in section headers and can be compared to those from the data directory for integrity.

2.2.2 System Service Dispatch Table. The system service dispatch table (SSDT) is used to hold addresses of Windows system call functions in order for the system service dispatcher to find and execute system services belonging to the kernel, `Ntoskrnl.exe`. Additionally, a shadow SSDT exists that holds addresses of Windows window manager and graphics device interface (GDI) services exported from `Win32k.sys` [RS04]. Figure 2.7 illustrates how a Windows system call is made from an application. The application calls the `Kernel32.dll` function, `WriteFile`, which calls the Windows specific `Ntdll.dll` subsystem function, `NtWriteFile`. Next, a software interrupt is generated using the `SYSENTER` or `SYSCALL` instruction on Intel or AMD architecture, respectively.¹ This prompts a transition to kernel mode where the service number argument passed from above is translated into the address of the `Ntoskrnl.exe` version of `NtWriteFile` using the SSDT. Finally, the `NtWriteFile` processes the service request and the call is complete. The right side of Figure 2.7 depicts a call to a GDI or window manager service and is similar to a kernel dispatch except that it does not involve `Kernel32.dll` because `Win32k.sys` is inherently Windows specific. Also, the shadow SSDT is used to look up the service function. The structure of this process makes it vulnerable to attack and is discussed in further detail in Section 2.3.2.2.

2.2.3 Process and Thread Lists. Processes and threads in Windows are represented by data structures called executive process and thread blocks, referred to as `_EPROCESS` and `_ETHREAD` in the kernel debugger. These structures contain information about each process or thread and pointers to other necessary data structures. These structures are linked together in memory by pointers called the flink and blink. Once a single process block is found, the list of processes can be traversed to enumerate all running processes on the system. Figure 2.8 shows how the lists are connected and related. The `ThreadListEntry` field of each process points to its list of threads. Detailed information about the `_EPROCESS` structure is contained in Appendix B.

¹`INT 0x2e` is used on processors prior to the Pentium II

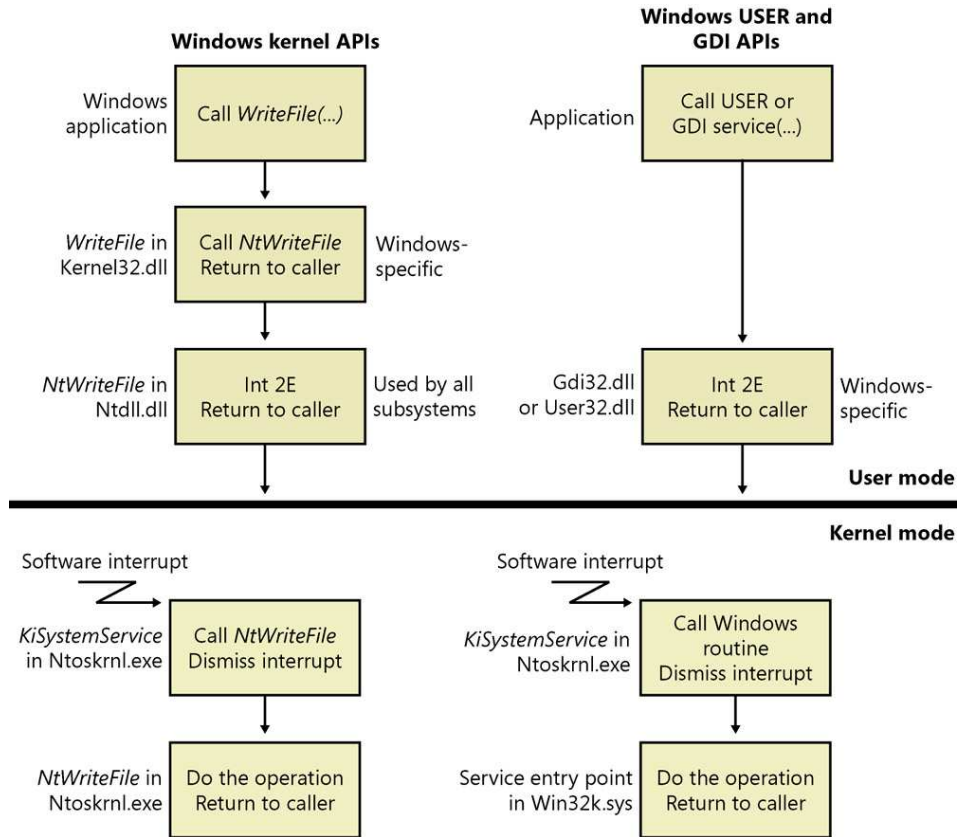


Figure 2.7: System Service Dispatching [RS04]

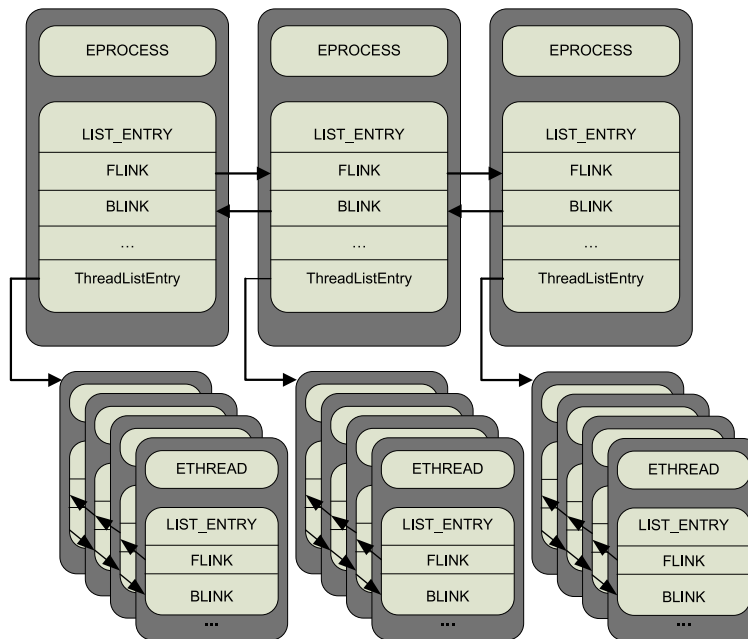


Figure 2.8: Process and Thread Lists

2.3 *Malware*

One major benefit of using virtualization for intrusion detection is that, excepting a flaw in the VMM itself, it isolates the VMM from malware running on a guest OS. The mechanisms in the hardware that control transitions from root to non-root provide an isolated execution environment for each VM that is running on the host system. Any communication in or out of an individual VM is handled by the VMM and subject to its controls. In order to understand the vulnerabilities of operating systems it is necessary to understand the different types of malware and methods used by malware to infect an operating system.

Figure 2.9 shows an adaptation of Rutkowska’s malware taxonomy [Rut06]. The figure shows the four different types of malware and where they reside with respect to other common memory structures. The large boxes represent processes with the largest one at the top being the system process, or kernel. The gray boxes represent the code and data sections of each process in memory with arrows representing hooks planted by malware to jump to the malware code. Hooking and its different implementation methods are discussed in Section 2.3.2. The black boxes represent malware and are classified based on their location.

Type 0 malware is a malicious program; acting on its own without help from or modification to the operating system or other processes. Type I malware modifies the static code sections of processes or the operating system, whereas type II malware modifies dynamic data sections. Finally, type III malware is introduced as malware that resides outside of any process or the operating system’s memory space [Rut06]. Malware Types 0, I, and II may be able to detect a VMM [GAWF07], but are not allowed to affect the VMM because of hardware protections that cause a VM exit and return control to the VMM. However, type III malware can affect the VMM by modifying the system BIOS or exploiting a bug in the VMM. Some type III malware exists, such as Bluepill and Vitriol [Rut07b, Zov06], but the millions of malware signatures that commercial antivirus programs use are for Type 0, I, or II malware [Sym08].

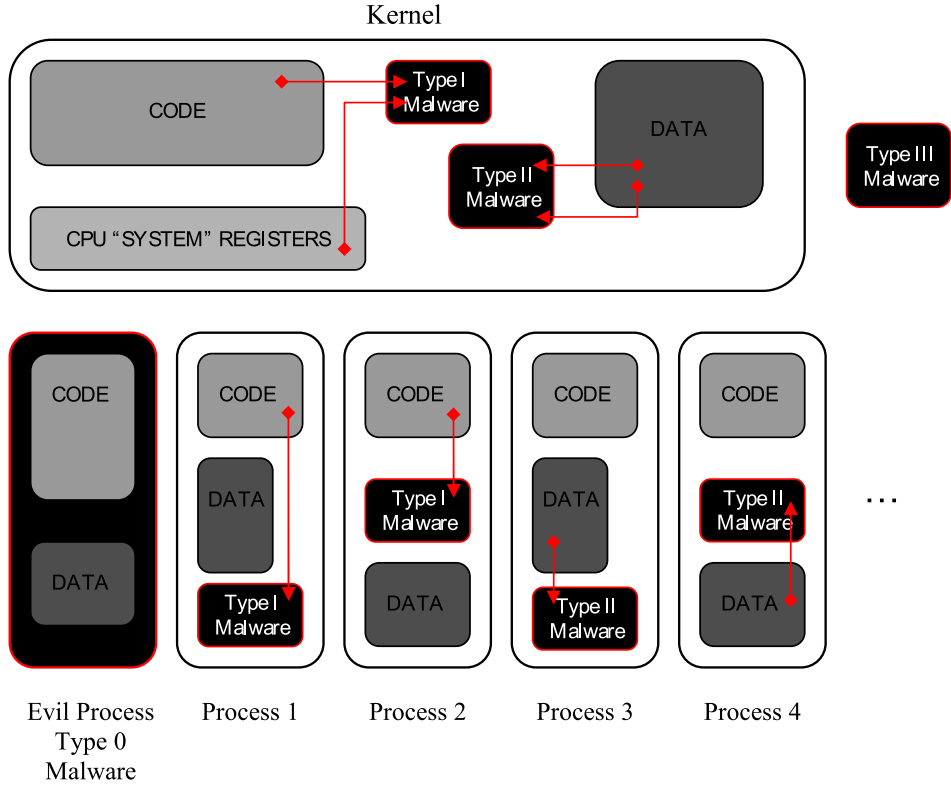


Figure 2.9: Malware Taxonomy adapted from Rutkowska [Rut06]

This means a VMM-based intrusion detection system is protected by hardware from most types of malware and can potentially detect malware of types 0, I, and II.

2.3.1 Rootkits. Rutkowska’s taxonomy classifies malware based on where they reside in the system whereas Skoudis [SZ03] takes a more functional approach dividing them into 8 categories based on function. Among those categories are user-level and kernel-level rootkits. Rootkits can be classified as type I or type II malware. Skoudis defines them as “Trojan horse backdoor tools that modify existing operating system software so that an attacker can keep access to and hide on a machine.” [SZ03] In other words, the user-level or kernel-level components that are used to carry out basic operating system functions are modified by rootkits for malicious purposes. Hooking and Direct Kernel Object Manipulation (DKOM) are used by rootkits to alter operating system function and are discussed in the following sections.

2.3.2 Hooking. One method that rootkits rely on to ensure their code is executed is by using hooks. In short, hooks are redirections of execution caused by replacing an address or piece of code within a process' memory space or an operating system data structure. Hooks can either be in user space or kernel space and allow a malicious piece of code to alter what is seen by the user.

2.3.2.1 User or Application Programming Interface Hooks. Application Programming Interface (API) hooks, though easier to detect than kernel hooks because they do not run in ring 0, can be useful. In order for malicious code to install API hooks, it must first have access to the process' memory space. This is typically accomplished using code or DLL injection [HB05]. Once attached to the process, the malicious code can modify the import address table (IAT), export address table (EAT), or install inline function hooks as described below:

- IAT and EAT Hooks

As described in the Microsoft PE and COFF specification, Windows executables and libraries must contain import and export information so that functions can be shared between them. At load time, any libraries that functions are being imported from are loaded into the process' memory along with the executable. During this process, the IAT of the executable is also loaded with the appropriate addresses for each imported library function. IAT and EAT hooking is accomplished by modifying the tables that hold the addresses of these imported or exported functions. When a hooked imported function is called, execution will be redirected to the hook function instead of the actual library function.

- Inline Hooks

Inline function hooking works by overwriting the first five bytes of a victim function with a jump to the address of the rootkit code. This is possible because most Windows functions after Windows XP SP2 contain a common five-byte preamble. The process is slightly more difficult with versions before SP2 that only contain a three-byte preamble, but still possible by simply saving and

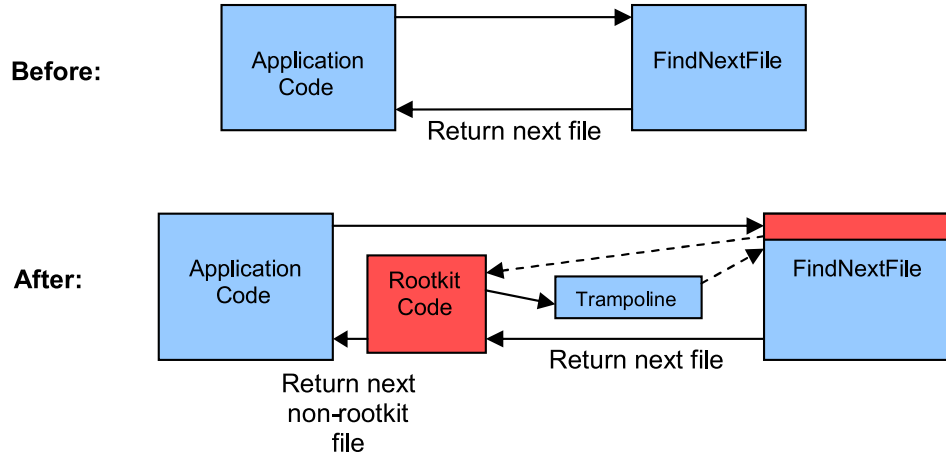


Figure 2.10: A User-Space Inline Hook Example [Rie06]

overwriting the two bytes following the preamble [HB05]. Listing II.1 shows the code bytes and assembly commands they represent for both preambles.

Listing II.1:

;SP2 and Later		Pre-SP2	
8bff	mov edi, edi	55	push ebp
55	push ebp	8bec	mov ebp, esp
8bec	mov ebp, esp		

The five bytes that are overwritten are saved and become part of a trampoline function that allows the victim function to execute and then returns execution to the rootkit code. At this time, the rootkit can modify the results returned by the victim function before they are returned to the calling application. Figure 2.10 gives a visual representation of the process.

2.3.2.2 Kernel Hooks.

- System Service Dispatch Table Hooks

SSDT hooking replaces the address of the system call functions in the SSDT with addresses pointing to a hook function. Figure 2.11 shows how a SSDT hook intercepts execution and is able to filter data returned by the system call. The

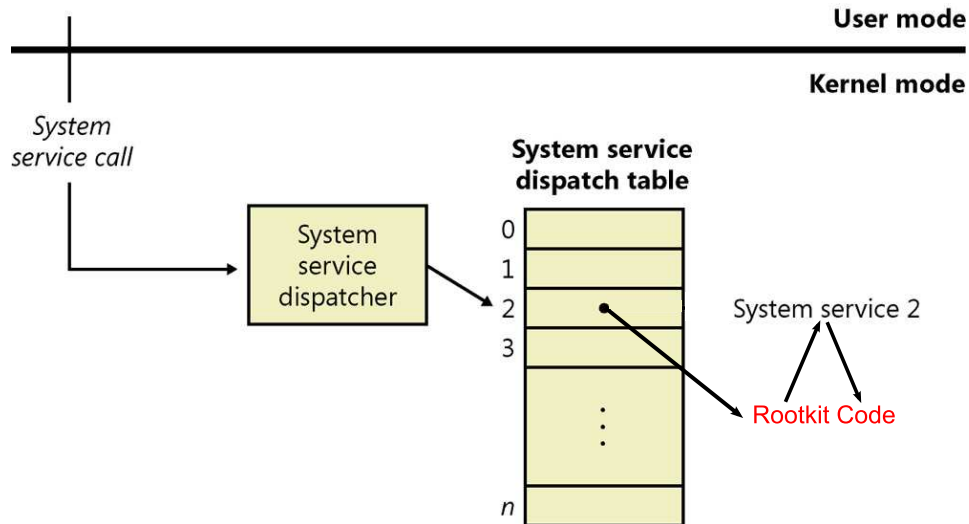


Figure 2.11: System Service Dispatch Table [RS04]

system service dispatcher uses the table to look up the appropriate system call address which has been replaced by a hook function address. The hook function can then call the real system service and filter results or simply return anything to the calling function. SSDT hooking is similar to IAT or EAT hooking in user-space except that the SSDT is a read-only kernel data structure. In order to modify values in the SSDT the rootkit has to bypass memory protections placed on the table. This can be done by clearing bit sixteen, the write protect bit, of the CR0 register or by creating a memory descriptor list with flags that allow writing to the region containing the SSDT [HB05]. In addition to their use by malware, SSDT hooks are used by many personal security products to enable operating system monitoring.

- Interrupt Descriptor Table (IDT) Hooks

As mentioned in Section 2.2.2, system calls on legacy hardware generate a software interrupt via the `INT 0x2e` instruction. Handling this interrupt requires looking at entry `0x2e` in the IDT for the address of that interrupt handler. Therefore, replacing the IDT entry at `0x2e` with the address of a hook function causes that function to run whenever any system call is made. Figure 2.7 can be used to illustrate the effect of an IDT hook. An IDT hook causes execution

to be redirected during the software interrupt preventing KiSystemService, the real INT 0x2e handler from being called. Each CPU in a system maintains an IDT and the address can be found by executing the SIDT instruction. Any entry in the IDT can be replaced allowing an attacker to intercept execution for events such as page faults in addition to system calls. One drawback to using IDT hooks is that execution does not return to the interrupt handler. In the case of hooking 0x2e, this means that execution does not return to the hook function and that it cannot be used for modifying data, only blocking it.

- **SYSENTER Hooks**

On Pentium II processors and above, the **SYSENTER** instruction is used to execute a fast system call. **SYSENTER** redirects execution to the address contained in the Model Specific Register (MSR), **IA32_SYSENTER_EIP** [Cor08c]. An attacker can read and write the value in this register from ring 0 using the **RDMSR** and **WRMSR** instructions. This enables an attacker that has obtained access to ring 0 to redirect execution to any arbitrary address.

- **Input/Output Request Packet (IRP) Function Table Hooks**

When a driver is loaded, a table containing the addresses of all the functions it uses to handle IRPs is initialized. IRPs hold information needed by the system to process I/O requests. A rootkit can replace entries in the IRP function table and control the results of these I/O requests. This hooking method is useful for intercepting network traffic as demonstrated by Hoglund [HB05].

2.3.3 Direct Kernel Object Manipulation. Using DKOM a rootkit can affect some of the same changes made by altering execution with hooking. DKOM relies on precise information about important operating system data structures. The Windows process list described in Section 2.2.3 is one example of a kernel object that is vulnerable to DKOM. A rootkit can use DKOM to change the Flink and Blink pointers in this list such that certain processes are unlinked from the list and will be hidden.

Additionally, DKOM can be used to hide device drivers, hide ports, elevate privileges and skew forensics [HB05].

2.4 Software Host-Based Intrusion Detection

The basic intrusion detection system (IDS) needs three components to function properly. The system first needs to use some sort of sensor to collect data. Second, the system has to classify or determine if the data obtained exhibits malicious activity. Finally, it needs to act on that determination and report an intrusion. These apply to all types of intrusion detection systems regardless of their design. Intrusion detection systems can be distributed or contained on a single machine, with the latter being the more prevalent [HFS98]. Network intrusion detection systems (NIDS) are used to monitor network traffic between multiple computers while host-based intrusion detection systems (HIDS) monitor multiple data on one specific machine, such as network traffic, operating system data structures, or system calls. This research focuses on host-based intrusion detection because of the advantages that can be afforded to it by virtualization.

2.4.1 Data Collection. When collecting data one must first determine what data is beneficial for intrusion detection. There has been much research on using system call traces as indicators of intrusion [HFS98, SYfZ⁺05, FS08, YA04, YXS⁺05, WD01]. System calls run at the highest privilege level in the kernel and therefore are a desirable target for attackers that want to gain privileged access to the system. System call monitoring simply observes the flow of system calls. In contrast, with system call interposition, execution can be redirected based on some criteria. For the purposes of this discussion, system call interposition and system call monitoring are used interchangeably because many of the same collection techniques apply whether interposing or simply monitoring the flow of system calls.

Several methods exist to intercept system calls. `Utrace`, `strace`, and the built-in `ptrace` are all Linux tools that can be used to capture system calls [McGb, McGa].

System call monitoring or interposition has also been implemented as a loadable Linux kernel module with Janus, BlueBox, and Systrace [Gar03, CC03, Pro03].

While system call interposition provides a useful indication of intrusion, it is also susceptible to mimicry and concurrency attacks [Pro03, Wat07]. Additionally, most research abstracts away the difficulty and overhead of interposing on system calls; these are discussed in more detail in Section 2.4.3. In addition to system calls, other OS data structures like those discussed in Section 2.2 along with files, directories, processes, and kernel-level modules can also be useful [JWX07].

2.4.2 Data Classification. Once data is obtained it needs to be classified as malicious or normal. This problem is the center of much research, though there are basically two techniques: signature-based (misuse-based) classification and anomaly-based classification. Signature-based classification requires the construction of a database of known malicious signatures. Anomaly-based classification establishes a baseline of behavior that is considered normal and uses statistical analysis to detect when behavior deviates from normal.

Signature-based detection is commonly used for antivirus software and network intrusion detection. These programs require significant maintenance to keep up with the release of hundreds of exploits and vulnerabilities released monthly [Sec08]. Symantec Antivirus contains definitions for 1,854,843 threats and risks [Sym08], and Snort, a popular open-source NIDS, has a database of approximately 13,000 rules for detecting intrusions [Sou].

A large body of research has been directed toward anomaly-based systems. Hofmeyr, et al. were the first to investigate system call anomalies for intrusion detection [HFS98], but many have followed using differing techniques. Many of these techniques were studied by Yasin [YA04] and are described here.

- N-Gram Sequence of N System Calls

This method, developed by Hofmeyr et al. examines sequences of system calls and compares them to a profile of “normal” behavior. The research showed that “normal” profiles were distinct for different applications and that it was possible to detect abnormal behavior using the sequences [HFS98].

- Finite State Automation Machine (FSA)

A FSA is created under normal process execution using the program counter (PC) from which the system call is made as each state and the system call name as the transition. Detection is accomplished by looking at the PC from which the call is made and determining if there is a matching transition from the current state to the new state [SBDB01]. One benefit of this approach over N-Gram is that it captures short and long-term correlations and the effects of loops and branches.

An alternate approach to the work of [SBDB01] that claims to address some of their limitations is proposed by Yu, et al [YXS⁺05]. Their approach uses the return address in the function stack as the state transition with no meaning assigned to the states. Similar to the FSA of Sekar, et al., if a particular transition is not present in the FSA an intrusion is detected. This method is immune to the recursive call and dynamically linked executable problems encountered by the N-Gram method.

- Static Analysis

This technique uses static analysis of system calls in program source code to model “normal” program behavior. Wagner and Dean [WD01] present four modeling methods to specify correct program behavior. Their models consist of a trivial model that establishes a white list of allowed system calls, a callgraph model based on the control flow of system calls, an abstract stack model that uses information from the system call stack, and a digraph model that is similar to the N-gram approach. Dynamic monitoring is used to determine if there is

a departure from normal program operation, as defined by the model. Wagner and Dean claim the callgraph model does not generate false alarms. However, the run-time overhead for their system varies widely from less than one second to greater than one hour [WD01].

- Virtual Path Model

The Virtual Path model is based on the interpretation of virtual paths between system calls. These paths consist of the return addresses of functions called in between the system calls. This model builds two data structures during training; the return address table and the virtual path table. The return address table contains entries for all return addresses contained in the call stacks of system calls. The virtual path table contains all virtual paths that are generated during normal execution. Detection is accomplished by extracting the current call stack and comparing return addresses to the return address table and virtual paths to the virtual path table. This model is also able to handle recursive calls and dynamically linked executables [FKF⁺03].

2.4.3 Host-Based IDS Limitations. There are many ways system call traces are used to detect intrusions, but their limitations are well documented [Gar03, Chu06]. One straightforward method to avoid detection is for an attacker to discover an allowed sequence of system calls that accomplishes his or her intent [WS02]. This “mimicry attack” takes advantage of the fact that many HIDS discard the parameters given to each system call and keep only the system call trace. As long as the attacker does not modify the sequence of system calls when inserting parameters the attack will remain undetected. Moreover, the permission level at which the system call interception runs in many implementations is a security issue. The system call traces provided by user-level tools are susceptible to modification by other processes or false reporting from a compromised kernel. Watson describes a concurrency vulnerability exploit for `sysjail`, an application based on the `Systrace` framework [Wat07]. Figure 2.12 shows how an attacker could take advantage of a concurrency vulnerability

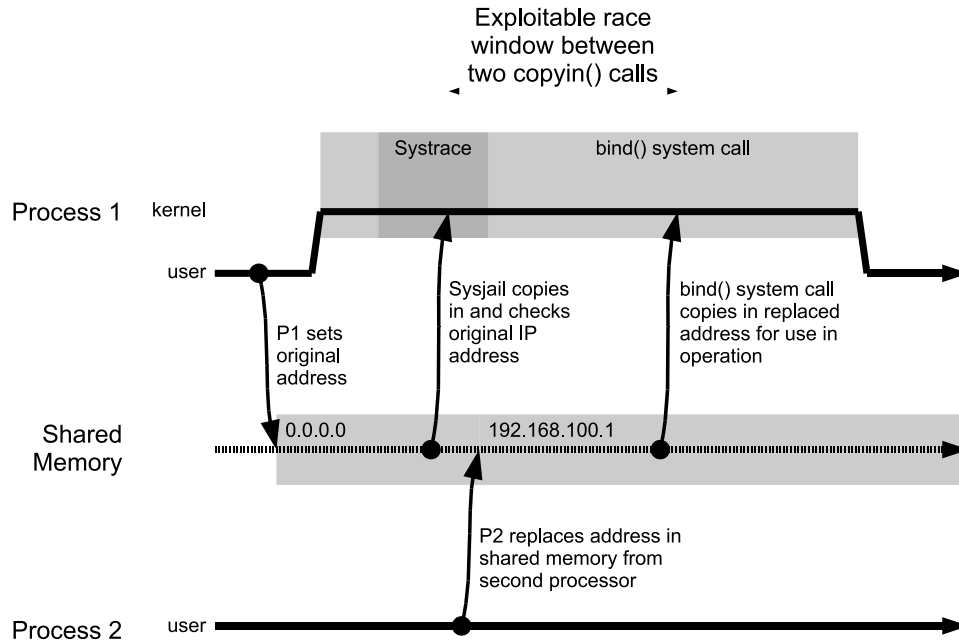


Figure 2.12: Race Exploit [Wat07]

and use a race condition to change the arguments passed to the system call after it has already been checked by the monitor. First, the original IP address is set by the user. Then, **Systrace** copies the address argument for system call monitoring. Next, before the system call executes the address argument is replaced by another user from another processor. Finally, the `bind()` system call copies the altered argument undetected by **Systrace**.

Many system-call-monitoring IDSes are “I/O-data-oblivious”, meaning they do not monitor I/O operations, which results in their vulnerability to persistent interposition attacks [PSJ08]. Persistent interposition attacks inject code using I/O operations without altering the control-flow or other system call arguments. While this stealth technique does not result in something as useful as a root shell, it does allow an attacker to modify data on the target without changing anything that would be detected by a system call monitoring IDS. Additionally, if the attacker managed to avoid detection while gaining access, the IDS could simply be turned off since it runs in the compromised kernel.

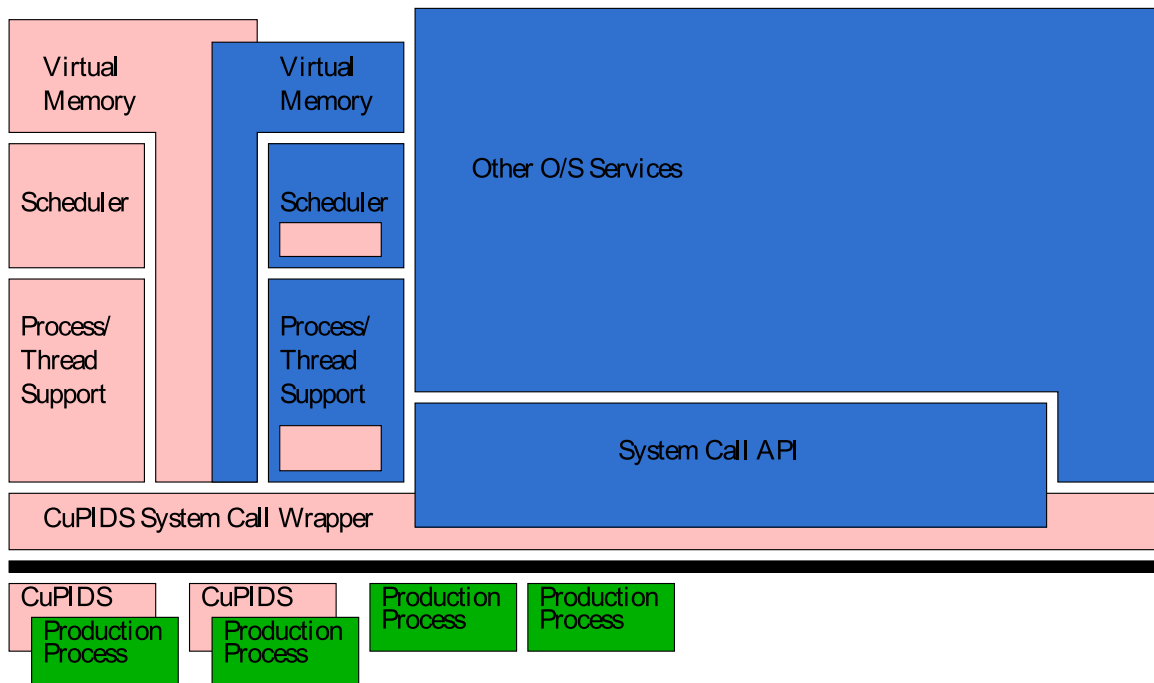
Finally, all of the system call monitoring HIDSs that have been presented are designed for Linux. Unfortunately, to port these to Windows would be a daunting task. Several issues make this problem more difficult in Windows [Chu06]:

- System call interfaces are undocumented and inconsistent
- Many Linux system call functions are provided from within user space in Windows
- Extensive use of dynamically linked libraries makes creation of “normal” difficult
- Intercepting thread context switching would be complex

2.5 Hardware Host-Based Intrusion Detection Methods

Hardware-based intrusion detection and integrity monitoring technologies have been investigated to support or replace software-based techniques. These typically involve the use of specialized hardware to interface or monitor some portion of the system. Using hardware makes the IDS much less vulnerable to any software-based attacks. However, the process of interfacing with the data that needs to be monitored adds to the complexity.

2.5.1 CuPIDS. The Co-Processor-Based Intrusion Detection System (CuPIDS) architecture is based on parallel monitoring of a production process on one processor by a shadow process on a separate processor [Wil05]. CuPIDS combines both software and hardware architecture components to achieve this monitoring. Figure 2.13 shows an overview of the CuPIDS software architecture. The items on the left edge of the figure are the CuPIDS monitors running on the shadow processor. The remaining elements represent production units running on the production processor. The overlap of the production and shadow units demonstrates the shadow monitor’s ability to monitor both passively and interactively the different components of the production units. The CuPIDS architecture includes several features that enable its monitoring capabilities. Among these are: secure inter-CuPIDS communication,



the ability to map virtual memory of a monitored process and interrupt and signal interception [Wil05].

The numerous parallel monitoring capabilities of CuPIDS mean that it is able to detect abnormalities within the system as they occur. Moreover, security is handled exclusively by the CuPIDS shadow process allowing for more complicated detection processes that might not be possible on a single-processor system. One drawback of CuPIDS is the loss of an entire CPU to security monitoring. Quad core processors are commonplace today and processors with more cores are on the horizon. If this architecture could be adapted to multi-core processors this loss could become much more tolerable. The communication between the production and shadow process is also of some concern because it relies on the kernel not being compromised. As suggested by Williams, an architecture similar to CuPIDS could be developed using a VMM or separate OS [Wil05].

2.5.2 Co-Pilot. Integrity checking via a coprocessor allows for an efficient means to verify host integrity while having minimal impact on host performance. Copilot, developed by Petroni, et al. [JFMA04], is a coprocessor-based Linux kernel integrity monitor implemented using a PCI add-in card. The system detects kernel level rootkits comparing known-good MD5 hashes of important kernel data structures with those that it reads at runtime.

Petroni et al. define the following conditions that need to be met to successfully monitor host system memory:

- Unrestricted access to the full range of host system’s main memory
- The monitor should be non-disruptive and invisible to the host system to the maximum degree possible
- The monitor should be completely independent from the host system
- The monitor should have sufficient processing power
- Provide the monitor with sufficient memory resources
- Utilize out-of-band reporting for secure communication with the administration station

Many of these requirements can be applied to a VMM-based IDS. Despite meeting all of their requirements for memory monitoring, the Copilot system still has some limitations. The system is only a passive monitor and cannot interpose on the functions of the host system. A byproduct of only having access to the host through direct memory access is that it cannot see kernel locks and may read data structures while they are being modified. This race condition could affect the ability of the system to monitor dynamic kernel data structures. Moreover, the current implementation only checks integrity every thirty seconds. This is more than enough time for an attack to cause damage to the system before it is detected [Mot07]. Also, the monitor is susceptible to relocation attacks that hide malicious code somewhere other than main memory, such as the cache. Finally, Rutkowska outlines an attack technique us-

ing memory-mapped I/O and the host system’s Northbridge that effectively disables PCI-based host memory acquisition like Copilot’s [Rut07a].

2.6 *Virtual Machine Introspection*

The isolation provided by virtualization increases the security of applications running in different virtual machines and the virtual machine monitor. The low level system access of the VMM has a side effect that makes monitoring and understanding operating system level abstractions difficult. This gap between the VMM and the internal operating system state, such as process information or disk structures, is referred to as the “semantic gap” [CN01].

Explicit and implicit information are two ways to overcome the semantic gap between VMM and OS. Explicit information is obtained with the cooperation of the guest OS. This requires modification of the kernel or detailed knowledge of the kernel data structures. This information is not always available or its structure may change from version to version [Jon07]. Moreover, this information cannot be trusted because a compromised guest OS could report incorrect or incomplete information. From the perspective of security, implicit information is more trustworthy because it is simply observed from outside of the VM instead of relying on the guest OS to report it. Implicit information is gained from analyzing architectural events as well as the raw memory and disk structures within the guest OS.

2.6.1 Explicit Information Introspection. Most methods used to extract explicit information make use of known structure and location of OS data structures. Information can also be obtained from debugging symbol information such as that in the system.map file in the Linux OS. Closed-source operating systems and structure differences from version to version can make it difficult to maintain programs that obtain data with this method. Despite this difficulty many systems make use of explicit information because of the accuracy and timeliness it provides.

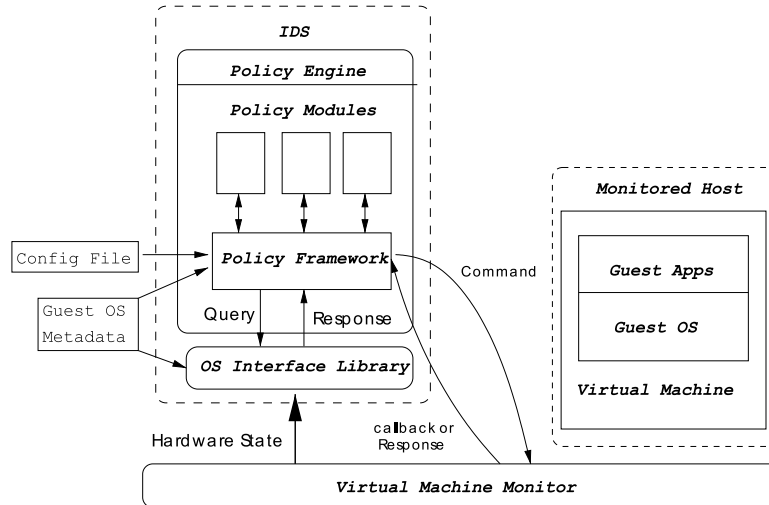


Figure 2.14: Livewire Architecture [GR03]

2.6.1.1 Livewire. Garfinkel and Rosenblum have developed Livewire, a VM-based intrusion detection system, that uses kernel debugging information to locate and read OS data structures [GR03]. Figure 2.14 illustrates the Livewire architecture. The architecture consists of a VMM interface, OS interface library, and a policy engine. The VMM controls the interaction of these three components.

In the case of Livewire, the OS interface library is what bridges the semantic gap. The VMM’s view of memory is similar to that given by `/dev/kmem` from within Linux. This allows Garfinkel and Rosenblum to use `crash`, a crash dump examination tool, to read OS data structures from within the VMM. `Crash` requires that the kernel is compiled with debugging information enabled so the OS data structures can be resolved from raw memory [Lin].

Livewire’s policy engine resides in a virtual machine and relies on the OS interface library and a policy framework to communicate with the monitored VM and the VMM respectively. Actions of the policy engine are based on several polling and event-driven policy modules. These modules determine when the policy engine will influence execution of the monitored VM. The policy framework controls the monitored VM via mechanisms provided by the VMM interface.

The VMM interface is composed of three types of commands:

- Inspection Commands - Allows the IDS to obtain VM state
- Monitor Commands - Allow event driven notification
- Administrative Commands - Allow IDS to control monitored VM

Garfinkel and Rosenblum acknowledge several weaknesses of the Livewire system. The weakness most relevant to this research is their discussion of fooling or compromising the OS interface library. The explicit information that the OS interface library relies on assumes that data structures in the monitored OS are consistent with a standard template. An attacker could modify the location of important kernel data structures so that they are inconsistent with the IDS information and therefore changes are undetectable. Moreover, a denial of service attack on the OS interface library is possible, though Livewire mitigates this problem by running the library in a separate process and monitoring its execution.

2.6.1.2 VMwatcher. A second VM-based introspection architecture based on *guest view casting* is VMwatcher, developed by Jiang et al. [JWX07]. The architecture runs the guest operating system on top of a VMM running in a host OS. Commercial malware detection utilities are run “out-of-the-box,” meaning they reside in the host OS rather than the guest OS being monitored [JWX07]. Figure 2.15 shows the VMwatcher architecture. *Guest view casting* is used to overcome the semantic gap by reconstructing the guest OS memory structures and virtual disk structures. The detection utilities are able to monitor the guest OS while being isolated in the host OS.

Semantic view reconstruction of both guest memory and virtual disks is accomplished using knowledge of their structure definitions and function semantics. File system structure is obtained from the virtual disks using the open source Linux device drivers and file system drivers. Guest OS data structures in memory are obtained by combining addresses from the OS symbol information and what is known about

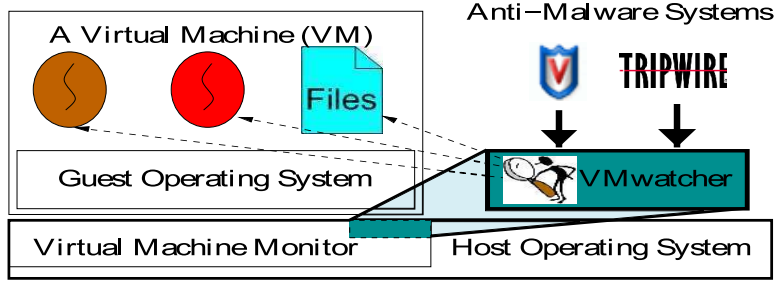


Figure 2.15: VMWatcher Architecture [JWX07]

the translation from physical to virtual address within the VM. The data structure assumptions made by VMwatcher make it vulnerable to the same relocation attacks as Livewire.

2.6.1.3 Lares. The Lares architecture is based on the Xen hypervisor with a specially designed virtual machine introspection library called XenAccess. In addition to a security monitoring VM, the Lares architecture adds hooks into the guest operating system to allow for active monitoring for malware or intrusions [PCSL08]. The guest OS is instrumented with jumps in strategic locations, such as program code, jump tables or other important structures. Figure 2.16 shows the Lares architecture. These hooks redirect execution to an embedded trampoline in the guest OS. This trampoline securely transfers information to the security monitoring VM via the hypervisor. Upon receiving information from the trampoline, the security VM uses an introspection API to obtain additional information about the state of the guest OS based on the needs of the security application. The hooks and trampoline are protected from malicious code in the guest OS by a memory protection component in the hypervisor. This component marks the memory location of hooks as read only causing a trap to the hypervisor if an access is attempted.

2.6.2 Implicit Information Introspection. Implicit information is gleaned from what is known about the system architecture. Interactions with specific structures in the architecture, such as registers, I/O subsystem or the memory management unit (MMU). This information does not depend on the OS and would be difficult for

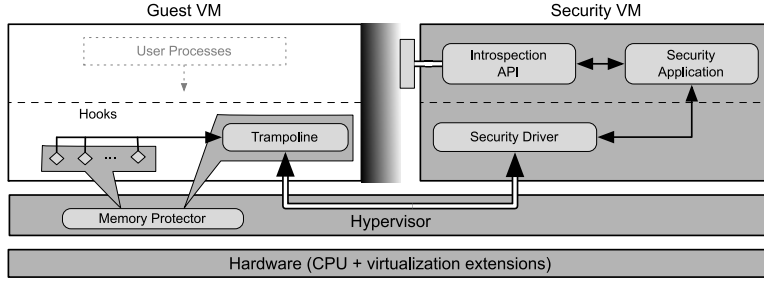


Figure 2.16: Lares Architecture [PCSL08]

a compromised OS to spoof. However, it is not without its limitations. Due to the methods used to obtain implicit information it is not likely that a VMM could produce all useful OS structures this way. Moreover, Jones demonstrated that implicit information sometimes suffers from delays and in some cases produces incorrect information [Jon07].

2.6.2.1 Antfarm and Lycosid. One source of implicit information is the x86 virtual memory architecture. A virtual address is translated to a physical address using page directories and page tables stored in memory. The system configuration register CR3 holds the physical address of the active address space’s page directory. Additionally, the most recently used page-directory and page table entries are stored in the translation lookaside buffers (TLB). Figure 2.17 shows how x86 virtual address translation occurs. The page directory address from the page directory base register, CR3, is used to locate the page directory. The most significant 10 bits of the linear address are used to index the page directory and locate a pointer to the appropriate page table. Linear address bits 21 through 12 are used to index the page table for the page table entry, which when combined with the 12 least significant bits of the linear address index the physical page.

CR3 is a privileged register and will cause an exit to the VMM if written to. Antfarm takes advantage of this to track processes in the guest virtual machine [Jon07]. It defines a unique address space identifier (ASID) that corresponds to a unique process. When a new page directory physical address is written to CR3 it is recorded in the

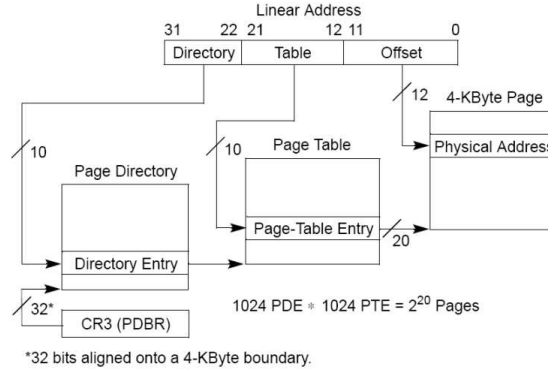


Figure 2.17: x86 Virtual Address Translation [Cor08a]

ASID registry as that process' ASID. Antfarm needs to observe three process-related events; process creation, context switch, and process exit.

- Process Creation and Context Switch

When the VMM detects that CR3 has been written to, the value is compared to those in the ASID registry. If the value is not present, then a new address space has been created and the ASID is added to the ASID registry. If it is already present, then a context switch has occurred and the new ASID is active.

- Process Exit

In order to isolate the different address spaces, the operating system must ensure that page directories and page tables are not reused without first being cleared of their values. Clearing of non-privileged page table entries is done on process exit by Windows and Linux. Tracking when the number of non-privileged page table entries reaches zero gives the VMM the first indication that a process has exited. Additionally, the TLB must be purged when an address space is deallocated. In the x86 architecture, this happens automatically when the value in CR3 is changed. Therefore, the VMM can infer that a process with the outgoing ASID has exited when there are zero non-privileged page table entries and the value in CR3 is changed [Jon07].

Jones uses Antfarm to implement the cross-view validation hidden process detector, Lycosid [Jon07]. The VMM-based detector uses Antfarm to obtain a trusted view of the system processes. Next, an untrusted view is obtained via a network connection to the VM and a user-level utility such as `ps` for Linux or `tasklist.exe` for Windows. These two different views are linked together by matching CPU time of each process. Finally, statistical techniques are used to increase the accuracy of detection in the face of interference and synchronization. Lycosid is vulnerable to desynchronization attacks, but this can be overcome by ensuring that any user-land process monitor was using the same view as Lycosid [Jon07].

2.7 Summary

This chapter presents fundamental concepts related to virtualization, host-based intrusion detection, and operating systems and recent research in those areas. First, virtualization methods are presented. Next, important Windows operating system data structures are explored in detail. Then, malware is defined and discussed. Later, principles of host-based intrusion detection are detailed. Finally, virtual machine introspection is explained.

III. Methodology

This chapter discusses the methodology used to measure the performance of the operating system and rootkit detection functions applied to Windows virtual machines. Section 3.1 defines the problem. Section 3.2 introduces the research goals. Section 3.3 presents the research approach. Section 3.4 discusses the experimental design for operating system detection and SSDT location and hook detection.

3.1 Problem Definition

The development of virtualization technology has provided a new method for isolated OS monitoring. Security monitoring can be performed from the VMM or a trusted VM using virtual machine introspection. Hardware virtualization extensions isolate the security monitor from the guest OS being monitored. However, the semantic gap must be bridged to accomplish this monitoring. Livewire, VMWatcher, Lares, and Antfarm utilize virtual machine introspection to monitor guest operating systems [GR03, JWX07, PCSL08, JADAD08]. Systems using explicit information (Livewire, VMWatcher, and Lares) require precompiled OS information for the guest operating system. Moreover, implicit information is limited and has only been used by Antfarm to monitor processes running in a guest OS. This research attempts to find a middle ground between explicit and implicit information introspection. Guest OS information is automatically obtained from guest memory or trusted sources and used in conjunction with the XenAccess library to monitor the guest's SSDT for hooks.

3.2 Goals

The methodology defined in this chapter is used to enable detection of SSDT hooks in a Windows guest domain from Xen's domain 0. Each of the goals defined below provide the means to carry out this detection:

- Detect the Guest Operating System
- Identify Version-Specific Offsets

- Locate OS Data Structures to Be Monitored

It is hypothesized that the Windows operating system version can be detected from domain 0 by examining the guest memory using forensic techniques. This version information can be used to retrieve the required offset information. Additionally, it is hypothesized that using techniques employed by malware writers [C01], the system service dispatch table can be located and monitored for hooks.

3.3 Approach

The hardware used for this research is a Dell D630 laptop with an Intel Core 2 Duo T7300 processor, two gigabytes of memory, and a 120 gigabyte hard drive. The T7300 processor is Intel Virtualization Technology capable and enables the use of hardware-assisted virtualization on the test platform.

The XenAccess library, version 0.4, is used to enable the virtual machine introspection necessary to monitor the Windows guest operating systems. The library provides the ability to view memory pages of one domain from another privileged domain [PCL07, Pay07]. It gives a raw view of guest memory that is much like that of a forensic memory dump. Therefore, many tools and techniques developed for forensic analysis can be applied to dynamically monitor guest operating systems from the management domain. This library is used because the availability of source code enables modification to suit the needs of this research. Introspection is accomplished by using a configuration file based on prior knowledge of the guest OS data structures. This research eliminates the need for a configuration file by using OS detection results and information obtained from guest memory to configure the XenAccess library. Compatibility with XenAccess drives the choice of many other portions of the testing environment and the core components are similar to [PCSL08]. The newest version of Xen that works with XenAccess 0.4 and Windows hardware-assisted virtual

Table 3.1: XenAccess Required Data Structure Information

XenAccess name	Windows Data Structure
win_tasks	EPROCESS->ActiveProcessLinks
win_pdbase	EPROCESS->Pcb->DirectoryTableBase
win_pid	EPROCESS->UniqueProcessId
win_peb	EPROCESS->Peb
win_iba	EPROCESS->Peb->ImageBaseAddress
win_ph	EPROCESS->Peb->ProcessHeap

machines is Xen version 3.1.4.¹ Additionally, Fedora 8 is used for the Xen domain 0, as it is the newest version of Fedora that runs as a Xen domain 0.

The Windows hardware-assisted virtual machines are created and configured as described in the example provided with the Xen source code, `xmexample.hvm`. A ten-gigabyte virtual disk is created for each guest domain, and the OS is installed from base version CD media and upgraded to each service pack version. Each guest OS is allotted 512 megabytes of memory and one virtual CPU. The setup and configuration of the test environment is described in detail in Appendix C.

3.4 *Experimental Design*

3.4.1 Operating System Detection. In order to realize the goal of version-independent, automatic Windows guest OS introspection, it is necessary to implement an OS detection function as part of this research. XenAccess requires knowledge of the location of several operating system data structures in order to be fully functional. For example, until the version-specific `EPROCESS->ActiveProcessLinks`, `EPROCESS->Pcb->DirectoryTableBase`, and `EPROCESS->UniqueProcessId` offsets are known, only the kernel memory can be mapped to dom0. Table 3.1 shows the data structures required by XenAccess for Windows guest domains. These offsets can change for different versions of Windows as well as with Windows service pack updates or hotfixes. This is the impetus for detecting the version of Windows that is running in a guest domain. The offset information for each version of Windows

¹XenAccess 0.5, released on January 5, 2009 claims support for Xen 3.3.0

can be obtained by looking at the `EPROCESS` structure in the Windows kernel debugger (`windbg`) using the `dt -v -b _EPROCESS` command. Appendix B shows the output of the `dt` command for Windows XP Service Pack 2 with the pertinent entries in boldface type². In addition to offsets, XenAccess requires a file containing the kernel exports for the particular version of windows. This file can be obtained by using the `dumpbin.exe` utility provided with Microsoft Visual Studio by executing `dumpbin.exe /exports ntoskrnl.exe`. Alternatively, `dumpbin.exe` comes with the free `MASM32` development environment. XenAccess requires the export information to be contained in a text file pointed to by the configuration file. XenAccess uses this file to lookup symbols from disk whenever the `windows_symbol_to_address()` function is called.

Determining the OS version begins with a list of known base addresses where the Windows kernel is loaded into memory. Like the offsets described above, this address can change between major versions of Windows as well as service packs and hotfixes. This information is also obtained from the kernel debugger. The technique used for OS detection is based on a Perl script used to analyze memory dumps by Harlan Carvey called `kern.pl` [Car07] [Car06]. It was chosen because export symbol information contained in the kernel executable can be obtained with minimal effort after the OS detection is complete. This completely automates the XenAccess configuration process by using the exports retrieved from memory and the known offsets for the detected version. For an added measure of security, the exports obtained from memory can be compared to a known set of export symbols obtained from an untainted kernel executable.

The OS detection algorithm works by starting at a known kernel base address and stepping through memory guided by the `PECOFF` and `VS_VERSION_INFO` formats as shown in Figure 3.1 [Cor08d] [Cor08e]. Arrows from one box to another indicate following a pointer, while the lines terminated by a dot indicate an offset into

²Output of the `dt` command for all OS versions used in this research are included in electronic form with the source code.

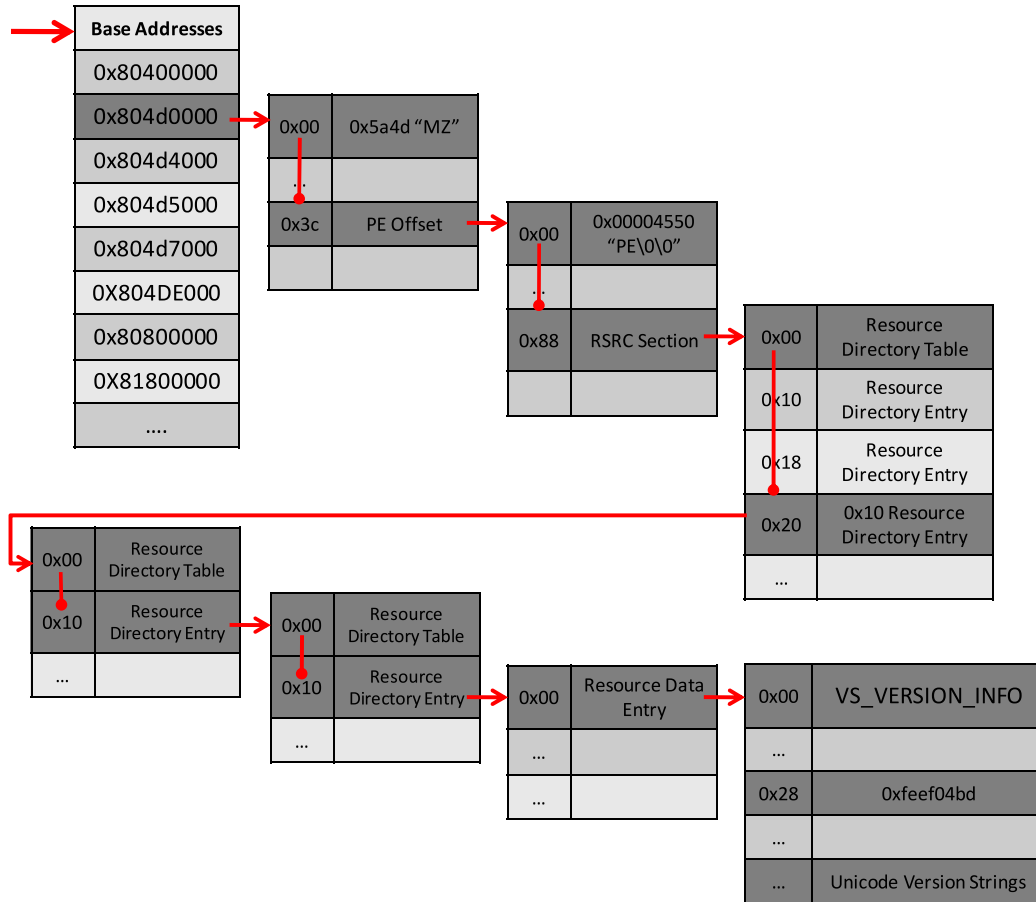


Figure 3.1: OS Detection Process

the same region of memory. If the `0xfeef04bd` signature in the `VS_VERSION_INFO` structure is found, a successful detection is reported, the kernel base address is set, and the version information strings are saved. If any signatures are not found or an attempt to map a specific memory page results in a null pointer, testing moves to the next base address.

After OS detection is finished, many pointers to the kernel executable still exist. This feature allows for the retrieval of the addresses of functions exported by the kernel. The kernel's optional header, where the address to the resource table is found, also contains the address of the kernel's export table. The kernel export area contains three parallel tables with the export names, addresses, and ordinals. In order to

put them in the name and address pair format desired by XenAccess, they must be reindexed using the pseudocode shown in Listing III.1.

Listing III.1:

```
i = Search_ExportNamePointerTable(ExportName);  
ordinal = ExportOrdinalTable[i];  
SymbolRVA = ExportAddressTable[ordinal - OrdinalBase];
```

With this step complete, the exports obtained from the kernel image in memory are stored in the XenAccess instance structure for later retrieval using `windows_symbol_to_address_from_mem()`, which replaces the standard library function, `windows_symbol_to_address()`.

3.4.1.1 Performance Tests. The operating system detection algorithm is evaluated using three experiments scripted from the Xen domain 0. First, the accuracy of the operating system function is tested. Next, the initialization time is compared to the standard library. Last, the performance of the new symbol lookup function is compared to the standard library function.

Operating System Detection Performance. The experiment consists of thirty trials for each of nine Windows versions and Ubuntu 8.10. Table 3.2 lists the OS versions tested and the kernel base address associated with that version. For the purposes of the XenAccess library, Windows versions with the same base address and data structure format can be processed identically. However, they can be distinguished using the contents of the `VS_VERSION_INFO` structure if further granularity is desired. OS detection is considered a success if the correct kernel base address is returned. The script that tests each OS is shown in Listing III.2, with `<WindowsVersion>` representing the names of the OSes in Table 3.2. The body of the loop consists of starting the VM, sleeping for 60 seconds while it boots³, running the `os-detect` program with output directed to a file, and destroying the VM. This process is repeated 30 times.

³60 seconds was chosen to accommodate the OS with the longest boot time, Windows Vista.

Table 3.2: Operating System Base Addresses

Version	Base Address
Windows 2000 Professional SP4	0x80400000
Windows XP Professional	0x804d0000
Windows XP Professional SP1	0x804d4000
Windows XP Professional SP1a	0x804d4000
Windows XP Professional SP2	0x804d7000
Windows XP Professional SP3	0x804d7000
Windows 2003 Server	0x804de000
Windows 2003 Server SP1	0x80800000
Windows Vista Business	0x81800000
Ubuntu Intrepid Ibex 8.10	0xc0000000

Listing III.2:

```

for (( i = 1; i <= 30; i++))
do
    xm create /etc/xen/<WindowsVersion>.hvm
    sleep 60
    ./os-detect <WindowsVersion>HVM > results/OSdetect/<WindowsVersion>-\$i
    xm destroy <WindowsVersion>HVM
done

```

XenAccess Initialization Time Performance. During initialization, the standard XenAccess library must read in the operating system information from the configuration file and scan for the kernel image in memory. In contrast, the modified version used for this research scans known kernel base addresses and then sets configuration information based on the OS detected. The initialization process used in this research eliminates any disk accesses and blind memory scanning and therefore is more efficient. For the modified library, initialization time also includes the time to process the kernel exports because they are read during initialization.

In order to test the performance of the standard XenAccess library initialization versus the modified version, the test program is instrumented to read the processor's time stamp counter before the library initialization function is called and immediately after. This is done using the `RDTSC` instruction, which on the Intel Core 2 Duo proces-

sor, provides a monotonically increasing unique value, or tick, that is used to calculate relative performance [Cor08b]. The code in Listing III.3 shows the `time_hack()` function that is used make call to `RDTSC`. The `volatile` keyword is used to ensure that the code runs when specified and is not relocated by the compiler during its optimizations. The initialization performance test consists of 1,000 trials using each library on each of the nine Windows versions in Table 3.2, for a total of 18,000 trials.

Listing III.3:

```
uint64_t time_hack()
{
    uint32_t lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return (uint64_t)hi << 32 | lo;
}
```

Symbol Lookup Performance. The `time_hack()` function is also used to test the performance of the new `windows_symbol_to_address_from_mem()` function versus the standard library function `windows_symbol_to_address()`. The respective initialization procedures for the standard library and the modified library are carried out before the first call to `time_hack()`. Since both libraries search the symbols linearly and the number of symbols varies with different versions, the first, middle, and last symbols were chosen as search queries. Each trial consisting of the three separately measured lookups was performed 1,000 times for each library. Therefore, a total of 6,000 lookups are performed for each of 9 operating system versions for a total of 54,000 measured symbol lookups.

3.4.2 SSDT Location and Hook Detection. As defined by Hofmeyr et al., a host-based intrusion detection system must perform three functions: data collection, data classification, and data reporting [HFS98]. The system developed for this research accomplishes data collection via the XenAccess library with modifications, data classification by checking kernel memory boundaries, and data reporting by identifying hooks and removing them if desired.

3.4.2.1 Data Collection with XenAccess. The first step in enabling XenAccess to perform data collection for the purposes of intrusion detection is to detect the OS version that is running. Once this is accomplished, as discussed in Section 3.4.1, the library is initialized and introspection is performed from the Xen domain 0. This provides a raw view of memory, akin to a memory dump. The locations of OS data structures need to be determined so that they can be monitored. The SSDT is chosen for this experiment because approximately fifty percent of rootkits in the wild use this technique [KS07, Rie06].

The address of the SSDT can be obtained using the kernel export, `KeServiceDescriptorTable` as described in [DPB99]. However, as discussed in Section 2.2.2, this does not contain information for services exported from `Win32k.sys`. This research determines the location of the SSDT and shadow SSDT by using a technique similar to Coldcrow [C01]. The information necessary to locate the SSDT is contained as part of the `ETHREAD` structure of a Windows GDI process. Figure 3.2 illustrates the steps taken to find the SSDT using bold arrows. First, the kernel export `PsiInitialSystemProcess` is used to gain entry into the doubly linked Windows process list. This provides a pointer to the `System` process, which is not a Windows GDI process. In Windows GDI processes, the `Win32Process` field in the `EPROCESS` structure will be non-null. Next, the list is traversed until the current `EPROCESS->Win32Process` field is not null. To get to the `ETHREAD` structure the pointer in the `EPROCESS->Pcb->ThreadListHead` field is followed to the `ETHREAD` structure. Finally, the `ETHREAD` structure contains the `DirectoryTable` field which points to the Shadow SSDT.

3.4.2.2 Data Classification. In order to classify a particular address in the SSDT as safe or malicious, the location and size of the kernel and `win32k.sys` in memory are needed. All addresses in the SSDT should point to an address in the kernel's memory space, and addresses in the shadow SSDT should point into the memory space of `Win32k.sys`. This information can be obtained from the kernel

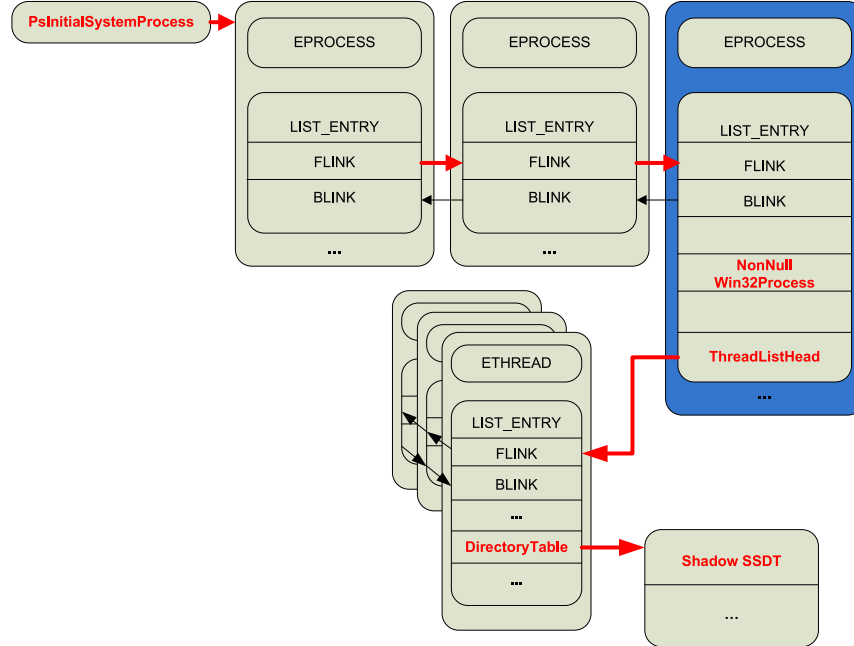


Figure 3.2: Shadow SSDT Location

module entry for each file in the Windows driver list as described by Hoglund [HB05]. This technique is adapted for use from outside the guest OS using XenAccess.

First a pointer to the module list must be obtained. The necessary kernel variable, **PsLoadedModuleList** is not exported by the kernel. For Windows XP and 2003⁴, it can be found in a structure pointed to by the **KdVersionBlock** field of the processor control region structure, **KPCR** [Bar, Ion]. In kernel mode, the **KPCR** is always located at **0xffdff000**. At offset **0x34**, the **KdVersionBlock** field points to the **_DBGKD_GET_VERSION64** structure. This structure is used by the kernel debugger and contains the kernel base address, the **PsLoadedModuleList** address, and a pointer to the **_KDDEBUGGER_DATA64** structure which contains many more unexported kernel variables. The format of these structures is listed in Appendix B.

Using **PsLoadedModuleList** the module list is traversed enumerating each module's memory space using the base address and size found in the module structure. If any address in the SSDT does not point to a location within the kernel's memory

⁴For Windows 2000 and Windows Vista **PsLoadedModuleList** is hardcoded in the library

space, it is classified as a hook and the module list is checked to see if it points to another module.

3.4.2.3 Data Reporting. The hook detector reports the system call number of any hook and the module that it points to, if applicable. The hook could point to a hidden module not in the module list or to a location that is not in any module's memory space. The system call number can be used to determine what function is being hooked using a system call table [Pro09]. This gives an indication about what the hook might be trying to achieve.

This research implements the SSDT monitor as a polling process from the Xen domain 0. The polling frequency is as fast as the processor will allow, approximately utilizing an entire core of the dual-core processor. For each iteration of the polling loop, the SSDT is located and checked for hooks. Hooks are automatically replaced with a known good value obtained before infection. The hooked system call number, target address, and kernel module's address space the target falls in are recorded in the result file.

3.4.2.4 Hook Detection Performance. To test the ability of the hook detector to locate hooks in the SSDT, two malware programs from the wild are used. Both the Agony rootkit and ProAgent spy tool are known to install hooks into the SSDT and they can be made to persist between restarts of the operating system. The malware is installed on VMs for each of the nine Windows versions. Shown in Listing III.4, the same script used to run the OS detection experiment is modified to run the check-ssdt program 30 times for each version of Windows.

Table 3.3: Agony Rootkit Options and Hooked System Calls

Option	System Call Hooked
-p <process name>	NtQuerySystemInformation
-f <file/directory name>	NtQueryDirectoryFile
-k <registry key>	NtEnumerateKey
-v <registry value>	NtEnumerateValueKey
-tcp <port>	NtDeviceIoControlFile
-udp <port>	NtDeviceIoControlFile
-space	NtQueryDirectoryFile
<drive letter>:<space to hide>	NtQueryVolumeInformationFile

Listing III.4:

```

for (( i = 1; i <= 30; i++))
do
    xm create /etc/xen/<WindowsVersion>.hvm
    sleep 60
    ./check-ssdt <WindowsVersion>HVM > results/<malware>/<WindowsVersion>-\$i
    xm destroy <WindowsVersion>HVM
done

```

Agony Rootkit. Agony is an open source, kernel-mode rootkit that hides processes, files, directories, registry keys and values, TCP and UDP ports, Windows services, and falsifies disk space. It is obtained from [Int06] and compiled from source using Dev-C++. Source code analysis reveals that Agony uses SSDT hooks to accomplish all hiding functions except service hiding. Agony is a command line tool with options allowing the user to specify what to hide. Table 3.3 lists the hiding options with the appropriate system call that is hooked by that option. For this experiment the following command line is used:

```

agony -r -p notepad.exe -f agony.c -k test -v test \
    -tcp 135 -udp 135 -space c:10

```

This results in the hooking of all six system calls listed in Table 3.3 and persists upon restart due to the -r option.

ProAgent. The ProAgent spy tool is a rootkit that collects password and user information on the infected system and emails it to the attacker. The program installs a driver named `JiurlPortHide.sys` onto the system. This driver is used to hide the port that the rootkit uses to send the collected information. Source code for `JiurlPortHide.sys` obtained from [Jiu08], reveals that this driver hooks the `NtDeviceIoControlFile` system call and persists through system restarts using system registry entries.

3.5 Summary

This chapter explains the methodology used to evaluate the operating system detection algorithm and SSDT hook detector implemented in this research. The operating system detection algorithm is evaluated based on detection accuracy, initialization time, and kernel export symbol lookup time. The SSDT hook detector is evaluated based on the ability to detect hooks placed in the SSDT by two rootkits.

IV. Results and Analysis

This chapter presents and analyzes the experimental results. Section 4.1.1 analyzes the results of the operating system detection and symbol lookup experiments. Next, Section 4.1.2 presents the results of the rootkit hook detection work. Finally, Section 4.4 summarizes the overall results from this research.

4.1 *Results and Analysis of Experiments*

4.1.1 Operating System Detection Analysis. The operating system detection experiment tests if the correct kernel base address is detected for each Windows version. Table 4.1 shows the outcome of the 30 trials for each operating system. The results show a positive detection for all versions of Windows.

Since no Linux OS detection is implemented, the Ubuntu Linux VM resulted in zero detections, as expected.

The Ubuntu Linux VM resulted in zero detections, as expected. False positive and false negative errors can be caused by malicious alteration of kernel memory. False negatives are easier to cause because only one piece of data along the detection algorithm's path in Figure 3.1 on page 39 needs to be changed to cause a negative result. Creating a false positive would require detailed knowledge of the detection algorithm and precisely crafted changes in kernel memory. Additionally, these changes may result in an unstable kernel.

4.1.1.1 Initialization Time. The standard XenAccess library must access the disk to read in the configuration file and scan memory to find the base of the kernel image. This results in a longer initialization time and additional effort to manually build a configuration file and kernel export file. A 1-sample t-test with a 95 percent confidence interval is applied to data obtained from the `time_hack()` function in the detection algorithm. The results of the t-tests for each Windows version are contained in Table 4.2. The table contains a row entry for the standard and modified libraries with each Windows version. The two entries for each Windows version, for the

Table 4.1: Operating System Detection Results

Version	Base Address	Successful Detections
Windows 2000 Professional SP4	0x80400000	30/30
Windows XP Professional	0x804d0000	30/30
Windows XP Professional SP1	0x804d4000	30/30
Windows XP Professional SP1a	0x804d4000	30/30
Windows XP Professional SP2	0x804d7000	30/30
Windows XP Professional SP3	0x804d7000	30/30
Windows 2003 Server	0x804de000	30/30
Windows 2003 Server SP1	0x80800000	30/30
Windows Vista Business	0x81800000	30/30
Ubuntu Intrepid Ibex 8.10	0xc0000000	0/30

modified library and standard library, are grouped using double lines for comparison. Each entry in the table shows the mean, standard deviation, standard error, and a 95% confidence interval for that Windows version. Figure 4.1 uses data from the t-test to compare the mean initialization time of the standard XenAccess library versus the modified version of the library for each operating system investigated. The confidence intervals for many of the versions are too small to be seen and show up as a dash. The columns greater than 10^9 CPU ticks represent versions of Windows for which the standard XenAccess library’s initial scanning method fails to find the kernel base address. At this failure point, execution time is already approximately 100 times greater than the modified library, so the trial is terminated. Therefore, actual initialization times are on the order of $4 * 10^{12}$ ticks, or about 40 minutes.

The failure of the initial scanning method may be a bug in XenAccess, in which case it is expected that the standard library initialization times would be similar for all Windows versions. Including the trials that are terminated early, the overall mean speedup is approximately 105. Including only those operating systems for which the standard XenAccess library’s initial scanning method succeeds, the overall mean speedup is 1.9.

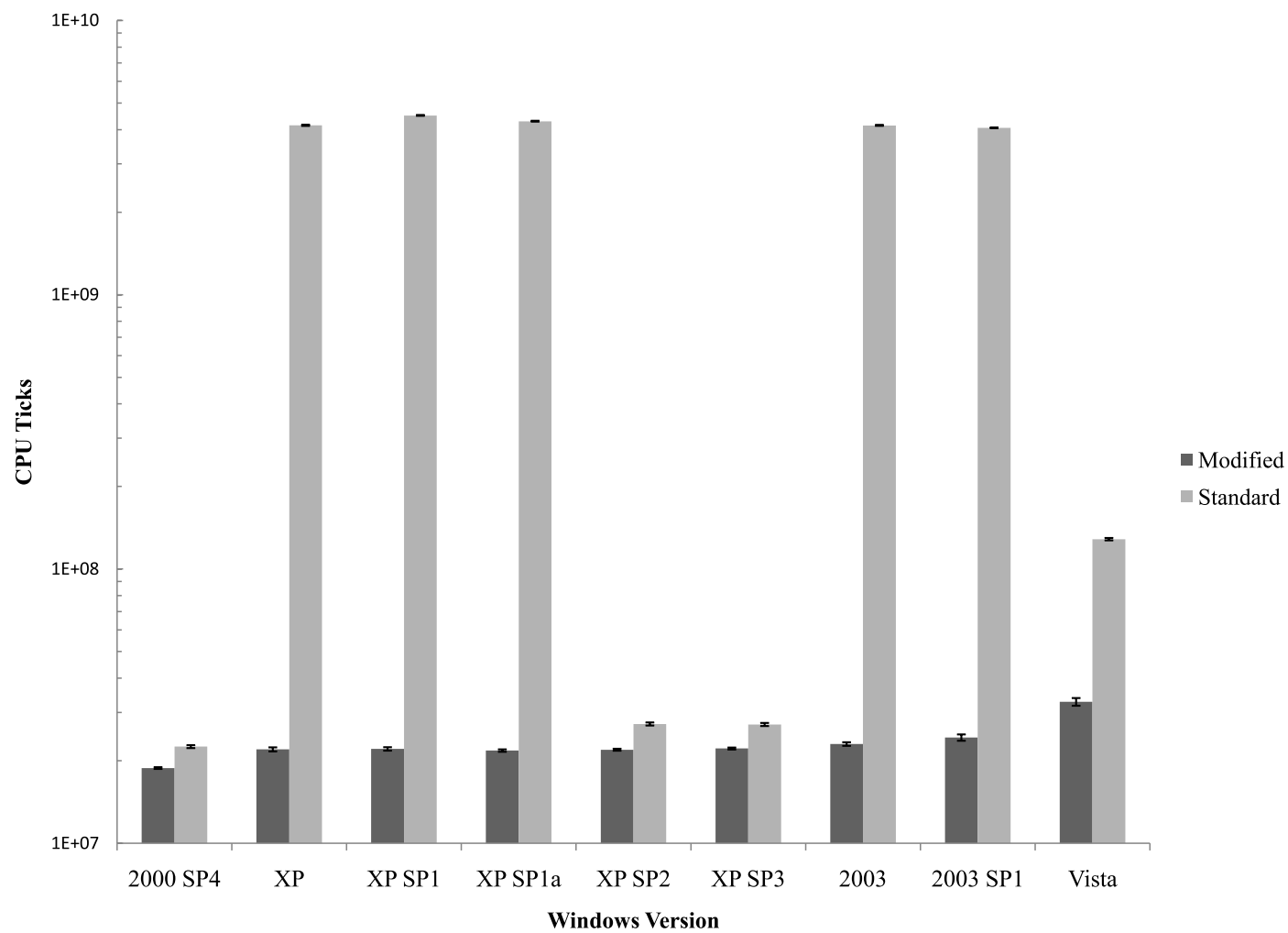


Figure 4.1: Mean Initialization Times - 95% Confidence Interval

Table 4.2: 1-sample t-Test for Initialization Time Comparison

Version	N	Mean	StDev	SE Mean	95% CI
Windows 2000 SP4 Mod	1000	18803414	2262363	71542	(18663024, 18943805)
Windows 2000 SP4 Std	1000	22517533	4517216	142847	(22237219, 22797847)
Windows XP Mod	1000	21996755	6042263	191073	(21621804, 22371706)
Windows XP Std	1000	4145416238	350270867	11076537	(4123680290, 4167152187)
Windows XP SP1 Mod	1000	22083674	5108116	161533	(21766691, 22400656)
Windows XP SP1 Std	1000	4504633660	251540793	7954418	(4489024375, 4520242945)
Windows XP SP1a Mod	1000	21775002	3572091	112959	(21553337, 21996667)
Windows XP SP1a Std	1000	4290522026	249610260	7893369	(4275032540, 4306011512)
Windows XP SP2 Mod	1000	21933356	2883767	91193	(21754404, 22112307)
Windows XP SP2 Std	1000	27226046	5697695	180177	(26872478, 27579615)
Windows XP SP3 Mod	1000	22152455	2954825	93440	(21969094, 22335815)
Windows XP SP3 Std	1000	27111457	5686774	179832	(26758566, 27464348)
Windows 2003 Mod	1000	23005312	5609042	177373	(22657245, 23353379)
Windows 2003 Std	1000	4143102517	271308223	8579519	(4126266570, 4159938463)
Windows 2003 SP1 Mod	1000	24293921	10416096	329386	(23647553, 24940288)
Windows 2003 SP1 Std	1000	4057345810	139567138	4413500	(4048685016, 4066006605)
Windows Vista Mod	1000	32786362	17041459	538898	(31728859, 33843864)
Windows Vista Std	1000	128445627	20999008	664047	(127142540, 129748714)

4.1.1.2 Symbol Lookup Time. The symbol lookup time is compared based on calls to the `windows_symbol_to_address()` function for the standard XenAccess library and calls to `windows_symbol_to_address_from_mem()` for the modified version. A 1-sample t-test with a 95 percent confidence interval is conducted for the data obtained from the symbol lookup time trials. The tabular data is contained in Appendix A. The means and confidence intervals are used to construct Figure 4.2 which shows the average number of CPU ticks taken to look up a symbol on each operating system with the standard and modified XenAccess libraries. Individual plots for each operating system and symbol lookup that include confidence intervals are found in Appendix A. On average, the library modified for this research is one order of magnitude faster than the standard library for all operating systems tested.

XenAccess maintains a 25-entry least recently used (LRU) cache for symbols. This means that after the initial lookup, subsequent lookups of the same symbol will not need to access the disk as long as it has not been evicted. Without a representative workload, it is difficult to calculate the miss rate of this cache. Because both versions of XenAccess benefit from this cache, the difference in lookup speed shown above is only valid for initial lookups.

In addition to an improvement in performance and eliminating the need for a file containing kernel exports, parsing the kernel exports from guest memory offers another avenue for intrusion detection. Crossview verification can be implemented by comparing the symbols from guest memory with symbols from a trusted source, such as an isolated Windows installation from trusted media or the Microsoft symbol server. The Microsoft symbol server provides debugging information for Windows operating system files. Any differences in the symbol information obtained from the untrusted and trusted sources indicate that the kernel has been tampered with.

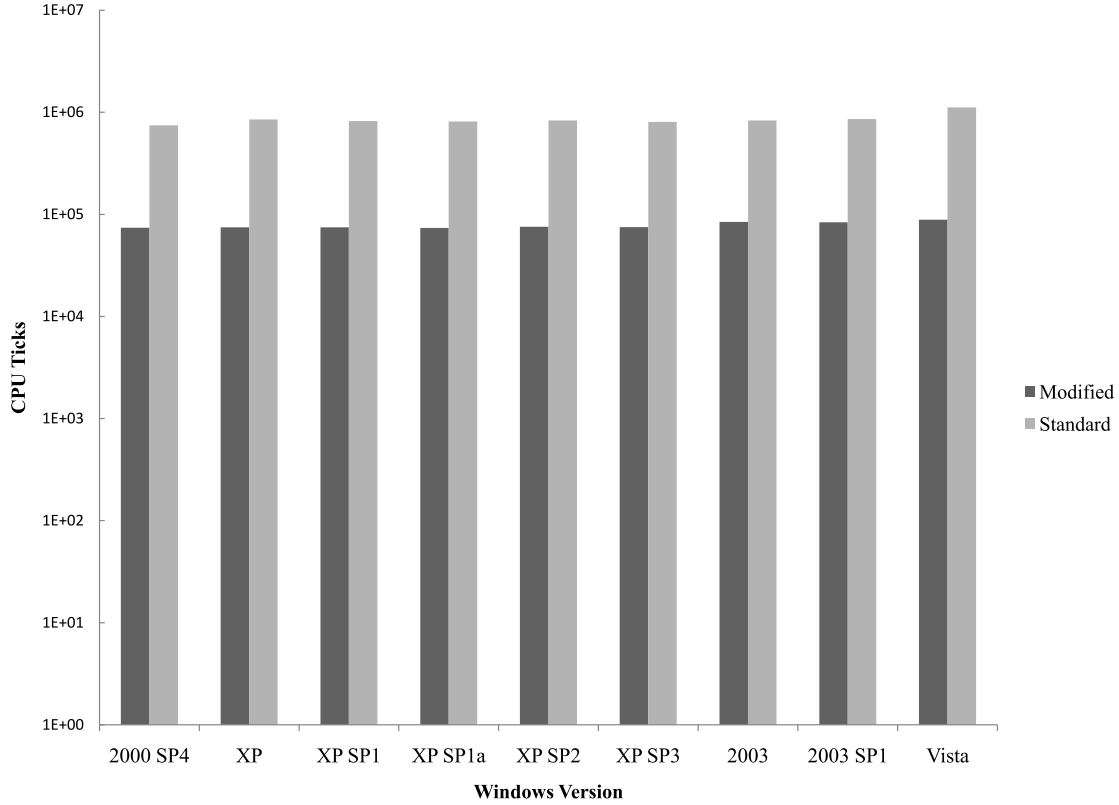


Figure 4.2: Average CPU Ticks for Kernel Symbol Lookup

4.1.2 SSDT Location and Hook Detection Analysis. Tables 4.3 and 4.4 show the results of the SSDT hook detection experiments. Each of the 30 trials produce identical results. The SSDT entry number reported by the detector is cross-referenced using the Metasploit Project system call table [Pro09] to obtain the name of the hooked system call. Note that the system call numbers only change with major versions of Windows so they are grouped together. The owning kernel module where the hook points to is reported by the detector and also listed in the tables.

The detector correctly reports the hooks installed by both rootkits. For Agony, the same six system calls are hooked in each version of Windows. This is the reason that only the system call numbers change in Table 4.3. ProAgent demonstrates the same behavior with all versions hooking `NtDeviceIoControlFile`.

Table 4.3: Hooks Detected for Agony Rootkit

Version	Entry #	System Call Name	Owning Module
Windows 2000 SP4	0x38	NtDeviceIoControlFile	agony.sys
	0x3c	NtEnumerateKey	agony.sys
	0x3d	NtEnumerateValueKey	agony.sys
	0x7d	NtQueryDirectoryFile	agony.sys
	0x97	NtQuerySystemInformation	agony.sys
	0x9d	NtQueryVolumeInformationFile	agony.sys
Windows XP, Windows XP SP1, Windows XP SP1a, Windows XP SP2, Windows XP SP3	0x42	NtDeviceIoControlFile	agony.sys
	0x47	NtEnumerateKey	agony.sys
	0x49	NtEnumerateValueKey	agony.sys
	0x91	NtQueryDirectoryFile	agony.sys
	0xad	NtQuerySystemInformation	agony.sys
	0xb3	NtQueryVolumeInformationFile	agony.sys
Windows 2003, Windows 2003 SP1	0x45	NtDeviceIoControlFile	agony.sys
	0x4b	NtEnumerateKey	agony.sys
	0x4d	NtEnumerateValueKey	agony.sys
	0x97	NtQueryDirectoryFile	agony.sys
	0xb5	NtQuerySystemInformation	agony.sys
	0xbb	NtQueryVolumeInformationFile	agony.sys
Windows Vista Business	0x7f	NtDeviceIoControlFile	agony.sys
	0x85	NtEnumerateKey	agony.sys
	0x88	NtEnumerateValueKey	agony.sys
	0xda	NtQueryDirectoryFile	agony.sys
	0xf8	NtQuerySystemInformation	agony.sys
	0xfe	NtQueryVolumeInformationFile	agony.sys

Table 4.4: Hooks Detected for ProAgent Rootkit

Version	Entry #	System Call Name	Owning Module
Windows 2000 SP4	0x38	NtDeviceIoControlFile	JiurlPortHide.sys
Windows XP	0x42	NtDeviceIoControlFile	JiurlPortHide.sys
Windows XP SP1	0x42	NtDeviceIoControlFile	JiurlPortHide.sys
Windows XP SP1a	0x42	NtDeviceIoControlFile	JiurlPortHide.sys
Windows XP SP2	0x42	NtDeviceIoControlFile	JiurlPortHide.sys
Windows XP SP3	0x42	NtDeviceIoControlFile	JiurlPortHide.sys
Windows 2003	0x45	NtDeviceIoControlFile	JiurlPortHide.sys
Windows 2003 SP1	0x45	NtDeviceIoControlFile	JiurlPortHide.sys
Windows Vista Business	0x7f	NtDeviceIoControlFile	JiurlPortHide.sys

4.2 Overall Analysis

This research successfully implements an operating system detection function to initialize the XenAccess library with appropriate configuration information. Additionally, export symbol information is obtained from memory eliminating the need for a symbol file and increasing initial symbol lookup speed by approximately 10 times. Moreover, initialization with the modified library achieves a speedup of 1.9 over the standard library.

The hook detection software successfully retrieves the location of SSDT from guest memory and uses it to monitor for hooks. The hook detector removes hooks placed in the SSDT automatically and replaces them with values obtained before hooking. The Agony and ProAgent rootkits are used to install hooks in the SSDT and are successfully detected by the monitor.

4.3 Limitations

The hook detector implemented in this research is a proof of concept for automated location of the SSDT from the Xen management domain. Many other OS data structures can be targeted by malware. Additionally, the SSDT is hooked by many personal security products. These hooks will be reported just as with the rootkit hooks creating false positives. One solution to this problem is to create a whitelist of kernel modules that are allowed to install hooks.

4.4 Summary

This chapter analyzes the experimental results obtained from the operating system detection and hook detection experiments. The initialization time and symbol look time are compared for the standard and modified XenAccess library. Next, the ability of the hook detector to detect hooks placed in the SSDT by malware is evaluated. Finally, an overall analysis of the results is presented.

V. Conclusions

This chapter provides a summary of key findings of this research. Section 5.1 draws conclusions based on experimental results. Section 5.2 discusses the impact of this research. Section 5.3 gives recommendations for future research in this area.

5.1 *Research Conclusions*

The XenAccess library can be automatically configured and used to monitor the system service dispatch table of a Windows VM from the Xen management domain. Obstacles caused by the semantic gap are overcome by investigating guest OS kernel memory using XenAccess and applying forensic memory processing techniques as listed below.

- Detecting the Guest Operating System

A Windows operating system version detection algorithm used for analysis of memory dumps [Car06] is adapted for use with XenAccess and successfully detects nine recent versions of Windows from the Xen management domain. A speedup of 1.9, on average, is observed for XenAccess initialization and an initialization file is not required.

- Identifying Version-Specific Offsets

The Windows kernel exports are obtained during initialization of the modified library. This eliminates the need for a system map file. Version-specific offsets in the `EPROCESS` structure must still be obtained using the kernel debugger. However, they are integrated into the library code and automatically set for the appropriate version as part of the operating system detection function.

- Locating OS Data Structures to Be Monitored

The system service dispatch table is located by adapting a technique used by malware [C01], to parse it from the guest OS memory. Hooks are detected using the kernel memory space as a boundary and the kernel module list to discover

where they point. Hooks implanted in the SSDT by the Agony and ProAgent rootkits are detected on all nine Windows versions tested.

5.2 *Research Impact*

This research demonstrates the value of using forensic techniques with virtual machine introspection to conduct host-based intrusion detection. Moreover, it is the first to integrate operating system detection as a means to automate introspection library configuration and is a step toward OS independent virtual machine introspection. This technique combined with hardware-assisted virtualization is a step forward from current ring 0 intrusion detection software.

The XenAccess library provides a promising open-source virtual machine introspection platform. The OS detection and data structure monitoring implemented in this thesis extend the capabilities of XenAccess. Additionally, the library initialization is 1.9 times faster and initial symbol lookups are an order of magnitude faster. Moreover, XenAccess 0.5 includes the scanning of known operating system base addresses during initialization to speed the process [Pag08]. Ideas presented as future work will also enhance the XenAccess library and increase security for virtual machines running in Xen.

5.3 *Recommendations for Future Work*

5.3.1 Version Specific Operating System Offsets. Modifications made to the XenAccess library for operating system detection used in this research still require prior knowledge of OS data structures. It may be possible for future research to eliminate this need for prior knowledge by obtaining kernel data structure information directly from the Microsoft symbol server when needed. Signature information needed to retrieve kernel symbol information is contained within the kernel memory. Additionally, the `_KDDEBUGGER_DATA64` structure defined in Appendix B contains the `EPROCESS->Peb` and `EPROCESS->Pcb->DirectoryTableBase` offsets needed by XenAccess.

5.3.2 OS Data Structure Monitoring. This research implements a monitor for the SSDT. This functionality can be extended to other important operating system data structures. The shadow SSDT can be monitored for Windows GDI function hooks. Monitors for the interrupt descriptor table and driver IRP tables can also be implemented. Extending this research to detect in-line system call function hooks only requires checking the first 5 bytes of the function at every address in the SSDT for jump instruction opcodes. A similar approach can be used for in-line function hook elsewhere.

5.3.3 System Call Interposition. The Lares architecture, discussed in Section 2.6.1.3 could be used as a data collection mechanism for system call traces. Software host-based intrusion detection systems principles like those discussed in Section 2.4 could then be applied for intrusion detection with these traces.

5.3.4 Linux OS Detection. The OS detection algorithm used in this research is Windows specific. However, a Linux OS detection function would also ease the burden of library configuration. Implementing a reliable Linux OS detector may not be feasible due to the large number of different versions and the ability to create custom kernels.

5.4 Summary

This chapter presents the research conclusions. Research impact is discussed, and recommendations for future work are given.

Appendix A. Symbol Lookup Data

This Page Intentionally Left Blank

Table A.1: 1-Sample t-Test of Symbol Lookup Times - Standard Library

Variable	N	Mean	StDev	SE Mean	95% CI
Windows 2000 SP4 First Symbol	1000	102883	9471	300	(102295, 103471)
Windows 2000 SP4 Middle Symbol	1000	786362	1943459	61458	(665761, 906962)
Windows 2000 SP4 Last Symbol	1000	1350774	575127	18187	(1315085, 1386463)
Windows XP First Symbol	1000	105914	16877	534	(104867, 106962)
Windows XP Middle Symbol	1000	929719	2744273	86782	(759424, 1100014)
Windows XP Last Symbol	1000	1516830	655517	20729	(1476152, 1557508)
Windows XP SP1 First Symbol	1000	106086	20574	651	(104810, 107363)
Windows XP SP1 Middle Symbol	1000	774600	345322	10920	(753171, 796029)
Windows XP SP1 Last Symbol	1000	1577734	846350	26764	(1525214, 1630254)
Windows XP SP1a First Symbol	1000	125571	339413	10733	(104508, 146633)
Windows XP SP1a Middle Symbol	1000	781595	432523	13678	(754754, 808435)
Windows XP SP1a Last Symbol	1000	1525383	667492	21108	(1483962, 1566804)
Windows XP SP2 First Symbol	1000	119357	278664	8812	(102064, 136649)
Windows XP SP2 Middle Symbol	1000	806261	487225	15407	(776026, 836496)
Windows XP SP2 Last Symbol	1000	1571303	765028	24192	(1523829, 1618776)
Windows XP SP3 First Symbol	1000	106128	10943	346	(105449, 106807)
Windows XP SP3 Middle Symbol	1000	780530	282694	8940	(762987, 798072)
Windows XP SP3 Last Symbol	1000	1532426	618272	19551	(1494060, 1570793)
Windows 2003 First Symbol	1000	133269	405252	12815	(108121, 158416)
Windows 2003 Middle Symbol	1000	817704	439093	13885	(790456, 844952)
Windows 2003 Last Symbol	1000	1546842	490806	15521	(1516385, 1577299)
Windows 2003 SP1 First Symbol	1000	106804	15441	488	(105846, 107762)
Windows 2003 SP1 Middle Symbol	1000	831238	338085	10691	(810258, 852217)
Windows 2003 SP1 Last Symbol	1000	1639586	615861	19475	(1601369, 1677803)
Windows Vista First Symbol	1000	112401	178000	5629	(101355, 123447)
Windows Vista Middle Symbol	1000	1114950	533549	16872	(1081841, 1148060)
Windows Vista Last Symbol	1000	2115550	627884	19855	(2076587, 2154514)

Table A.2: 1-Sample t-Test of Symbol Lookup Times - Modified Library

Variable	N	Mean	StDev	SE Mean	95% CI
Windows 2000 SP4 First Symbol	1000	61734	8088	256	(61232, 62236)
Windows 2000 SP4 Middle Symbol	1000	75085	198847	6288	(62746, 87425)
Windows 2000 SP4 Last Symbol	1000	85485	12416	393	(84715, 86256)
Windows XP First Symbol	1000	54777	5986	189	(54406, 55149)
Windows XP Middle Symbol	1000	73366	17210	544	(72298, 74434)
Windows XP Last Symbol	1000	95312	44161	1397	(92571, 98052)
Windows XP SP1 First Symbol	1000	54226	12423	393	(53455, 54997)
Windows XP SP1 Middle Symbol	1000	78425	190746	6032	(66589, 90262)
Windows XP SP1 Last Symbol	1000	91308	4687	148	(91018, 91599)
Windows XP SP1a First Symbol	1000	53801	7905	250	(53310, 54291)
Windows XP SP1a Middle Symbol	1000	72410	3947	125	(72165, 72655)
Windows XP SP1a Last Symbol	1000	94274	65038	2057	(90238, 98310)
Windows XP SP2 First Symbol	1000	62021	199532	6310	(49639, 74403)
Windows XP SP2 Middle Symbol	1000	72840	4064	129	(72588, 73092)
Windows XP SP2 Last Symbol	1000	92291	8977	284	(91733, 92848)
Windows XP SP3 First Symbol	1000	54212	8854	280	(53663, 54762)
Windows XP SP3 Middle Symbol	1000	72616	4995	158	(72306, 72926)
Windows XP SP3 Last Symbol	1000	97976	196954	6228	(85754, 110198)
Windows 2003 First Symbol	1000	60731	199680	6314	(48340, 73123)
Windows 2003 Middle Symbol	1000	86005	273248	8641	(69049, 102961)
Windows 2003 Last Symbol	1000	106328	275337	8707	(89242, 123414)
Windows 2003 SP1 First Symbol	1000	63349	280264	8863	(45957, 80740)
Windows 2003 SP1 Middle Symbol	1000	83159	216395	6843	(69730, 96587)
Windows 2003 SP1 Last Symbol	1000	104531	202544	6405	(91963, 117100)
Windows Vista First Symbol	1000	56257	38061	1204	(53895, 58618)
Windows Vista Middle Symbol	1000	89037	224591	7102	(75100, 102974)
Windows Vista Last Symbol	1000	120575	307522	9725	(101492, 139658)

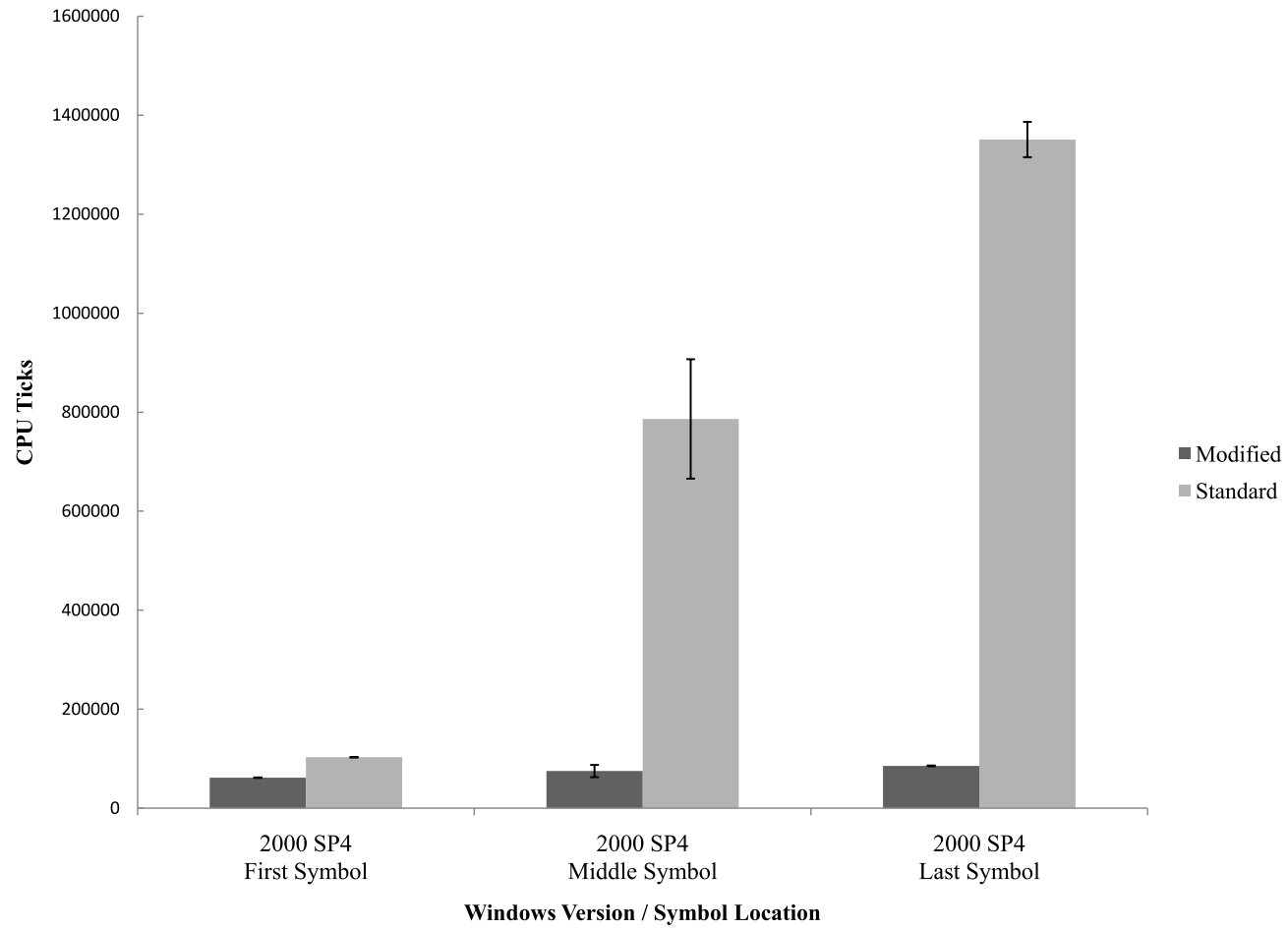


Figure A.1: Symbol Lookup Comparison for Windows 2000 SP4

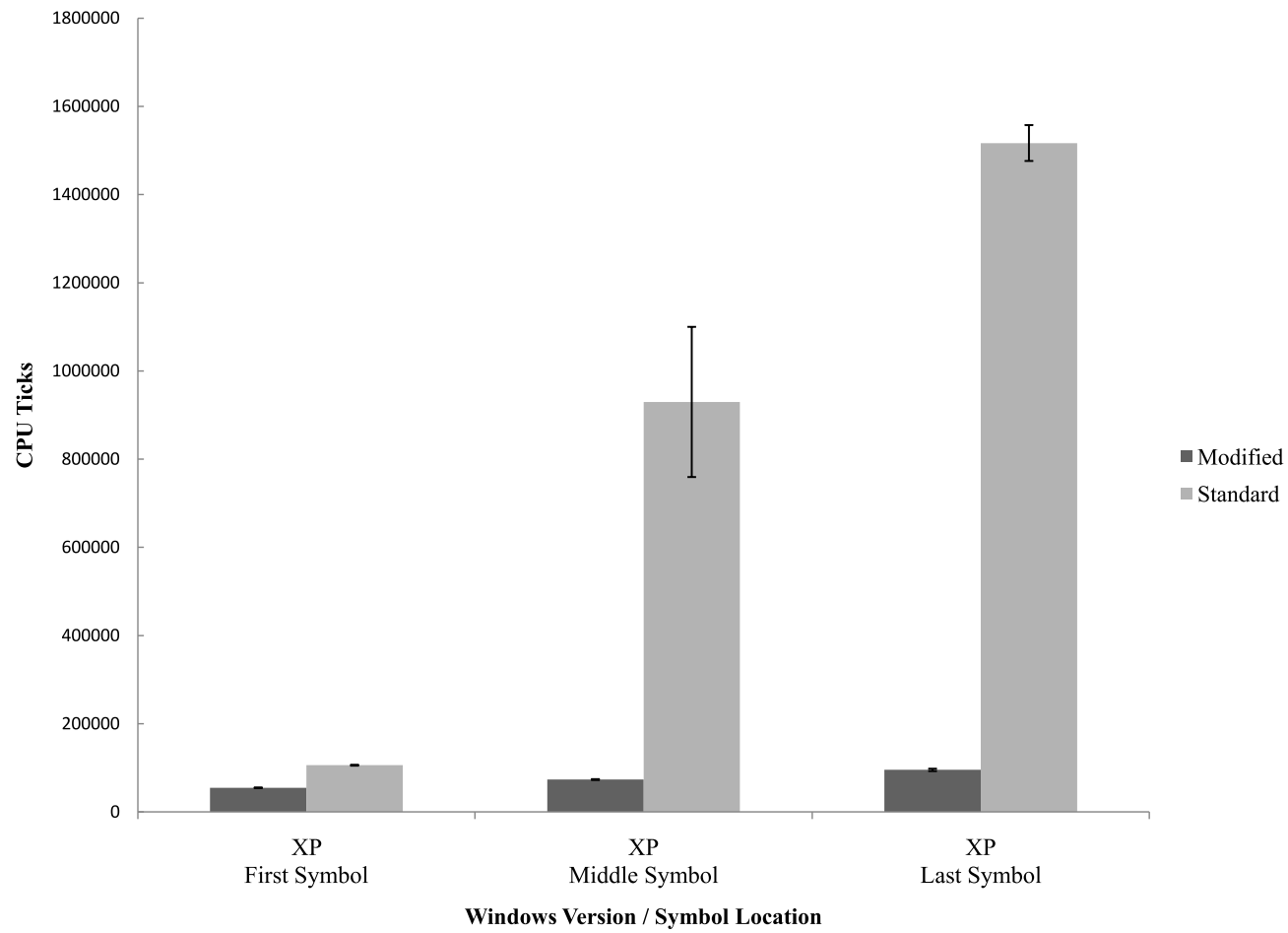


Figure A.2: Symbol Lookup Comparison for Windows XP

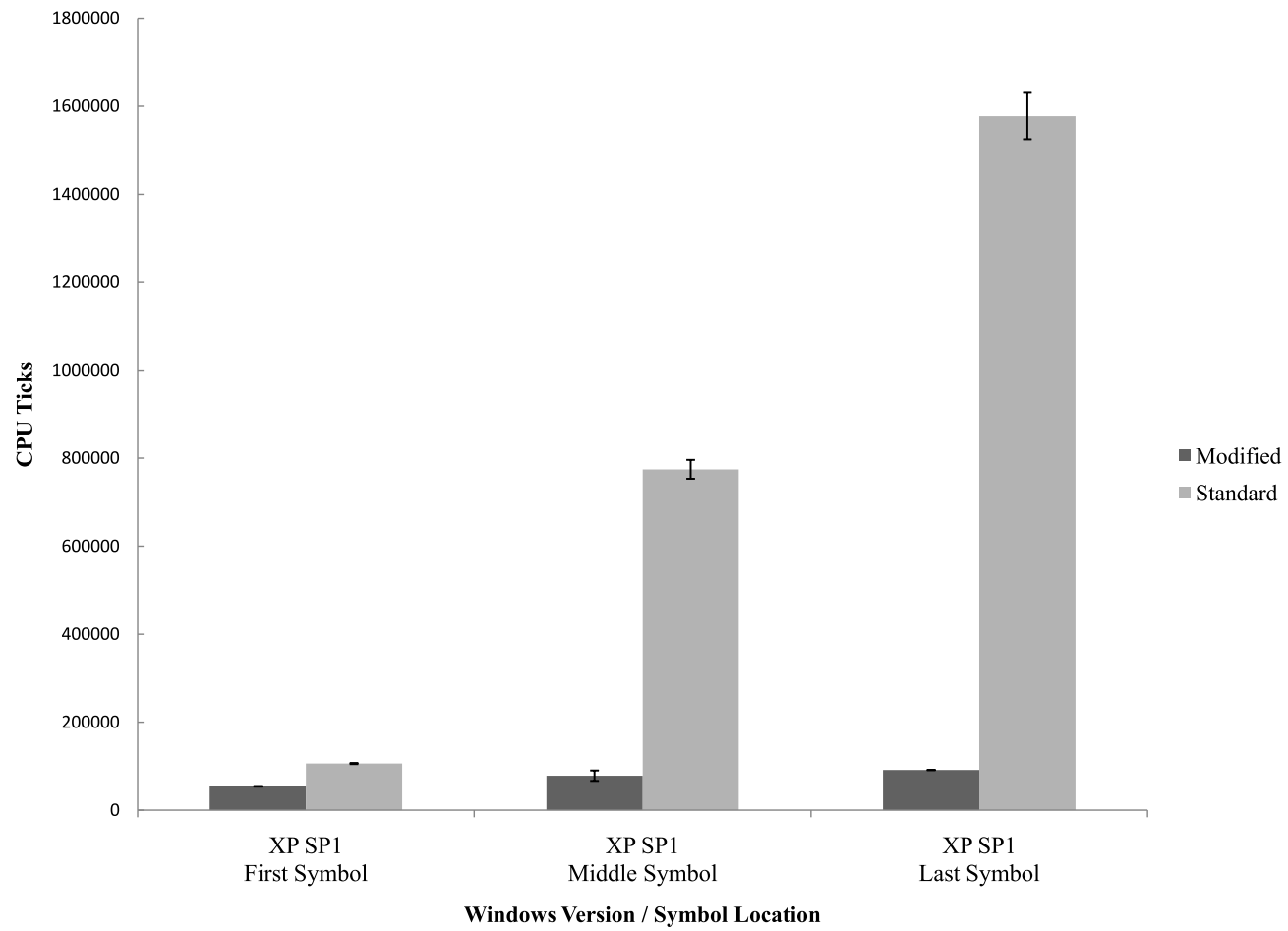


Figure A.3: Symbol Lookup Comparison for Windows XP SP1

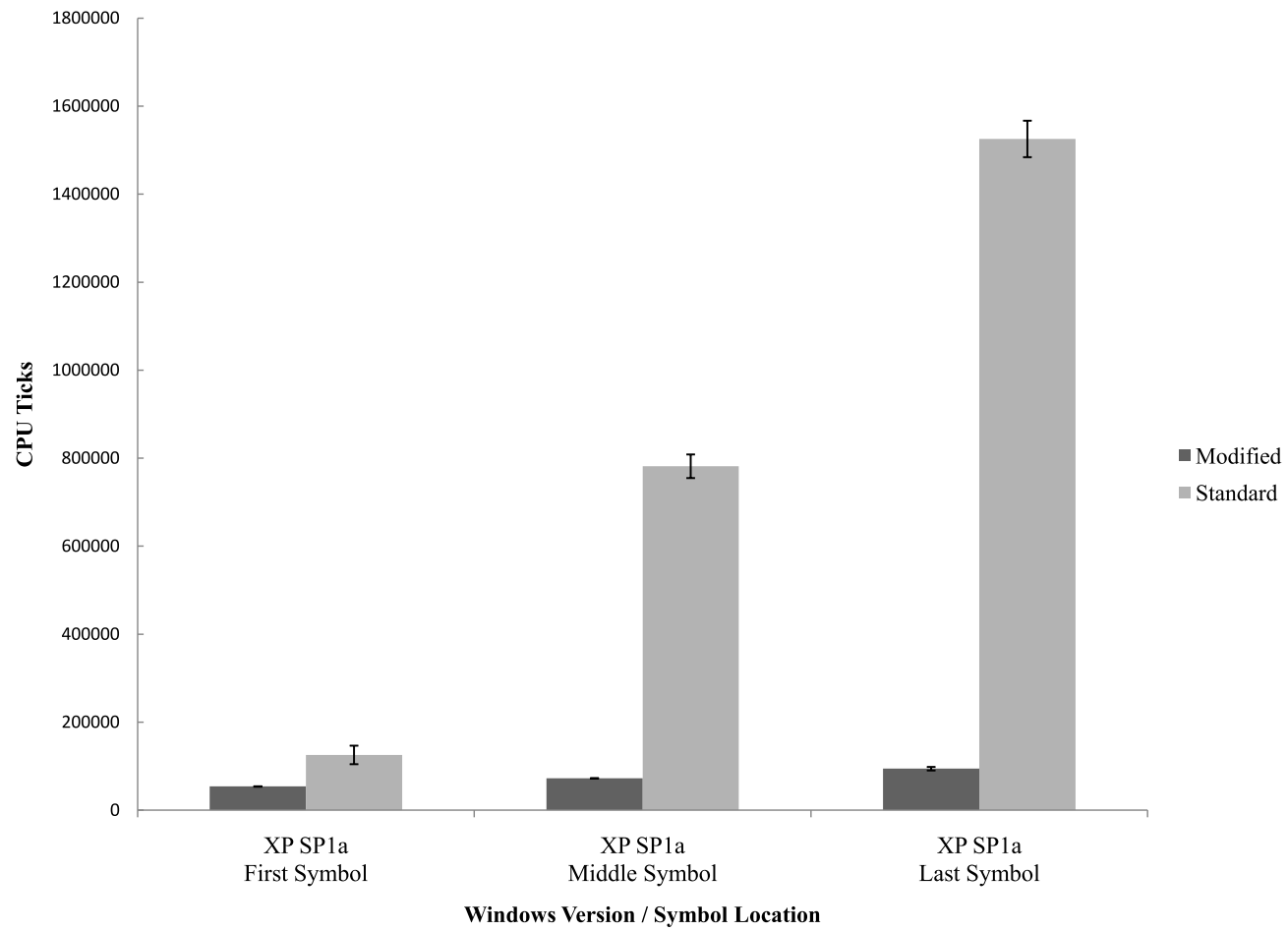


Figure A.4: Symbol Lookup Comparison for Windows XP SP1a

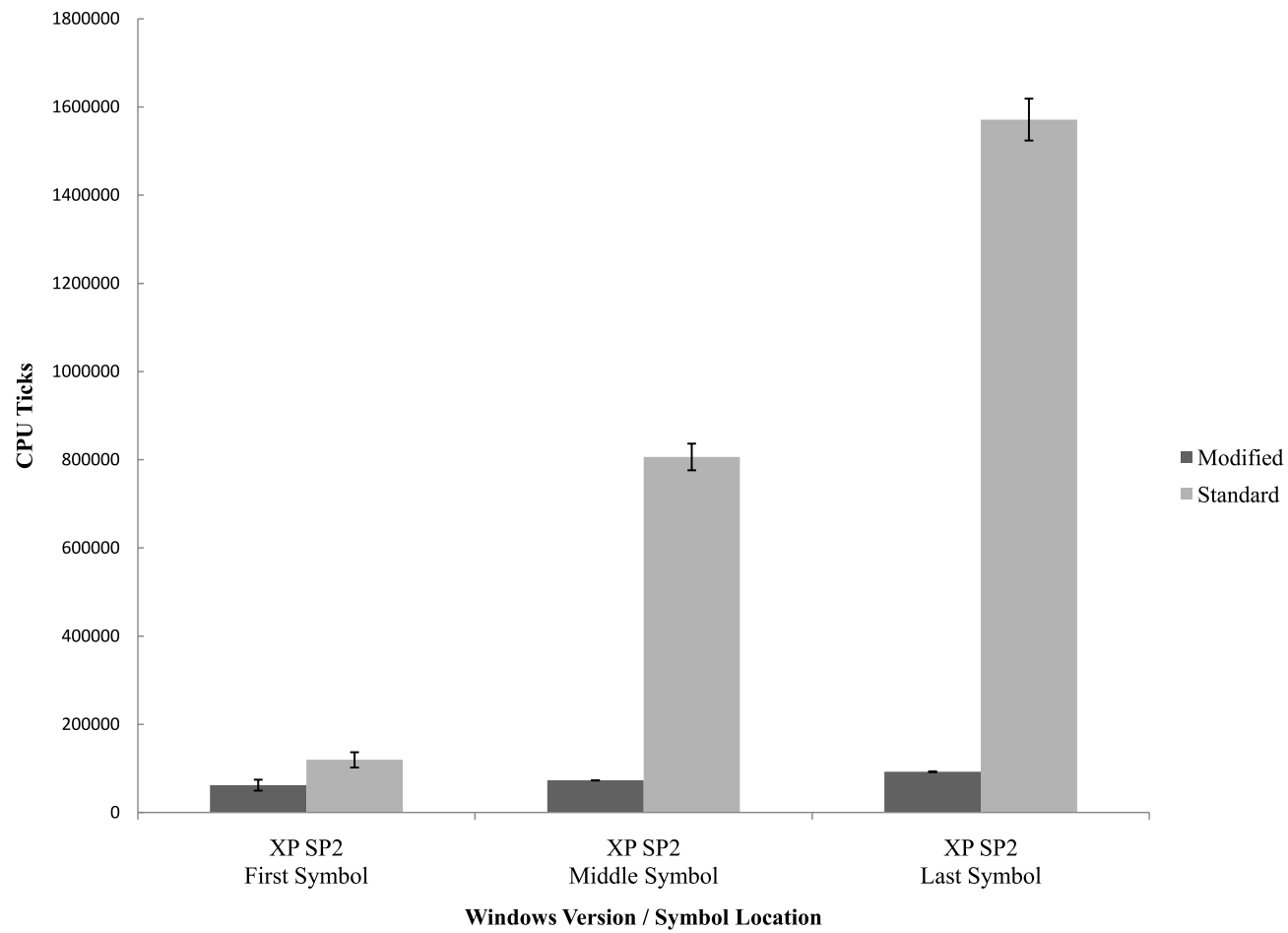


Figure A.5: Symbol Lookup Comparison for Windows XP SP2

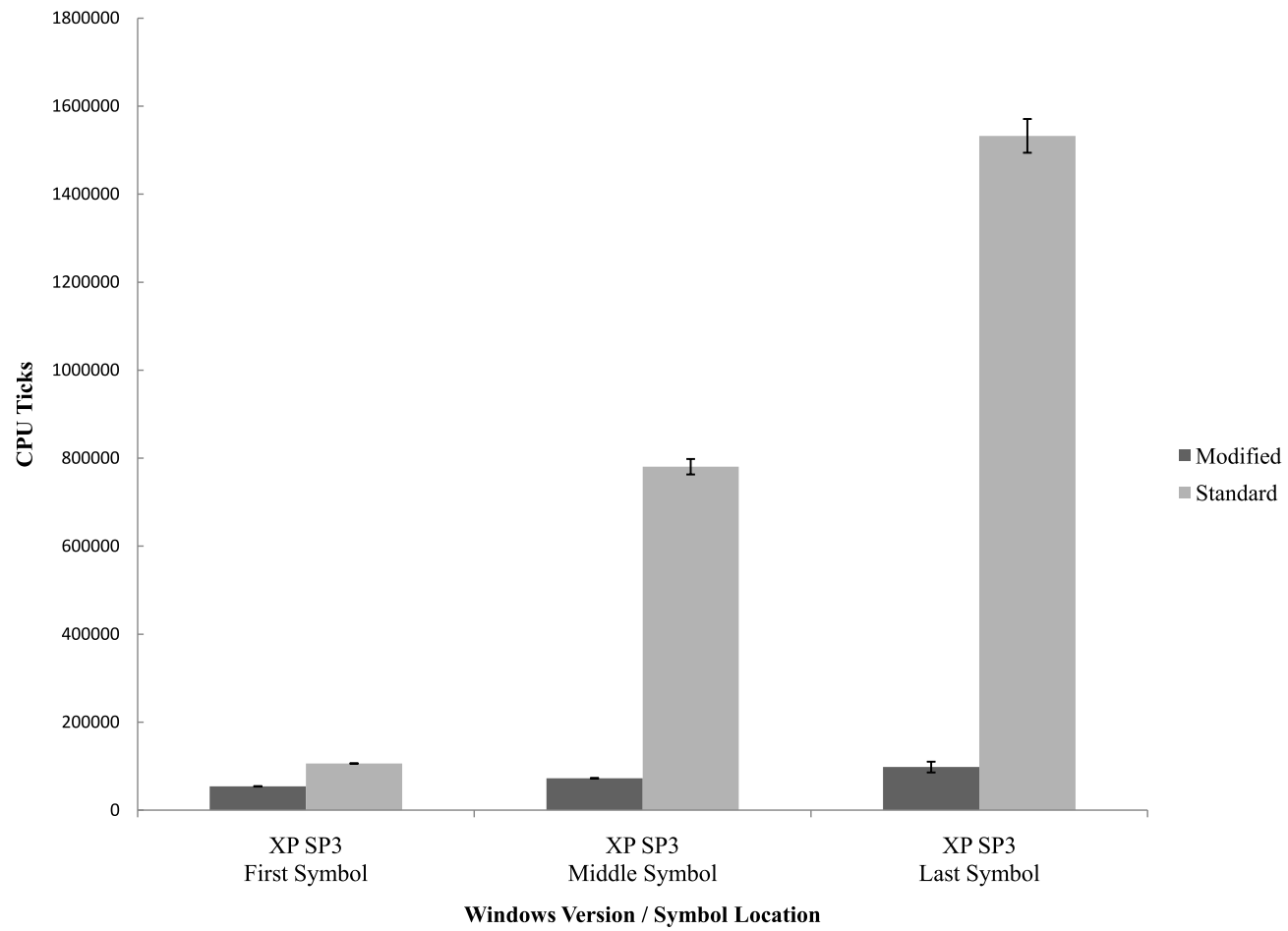


Figure A.6: Symbol Lookup Comparison for Windows XP SP3

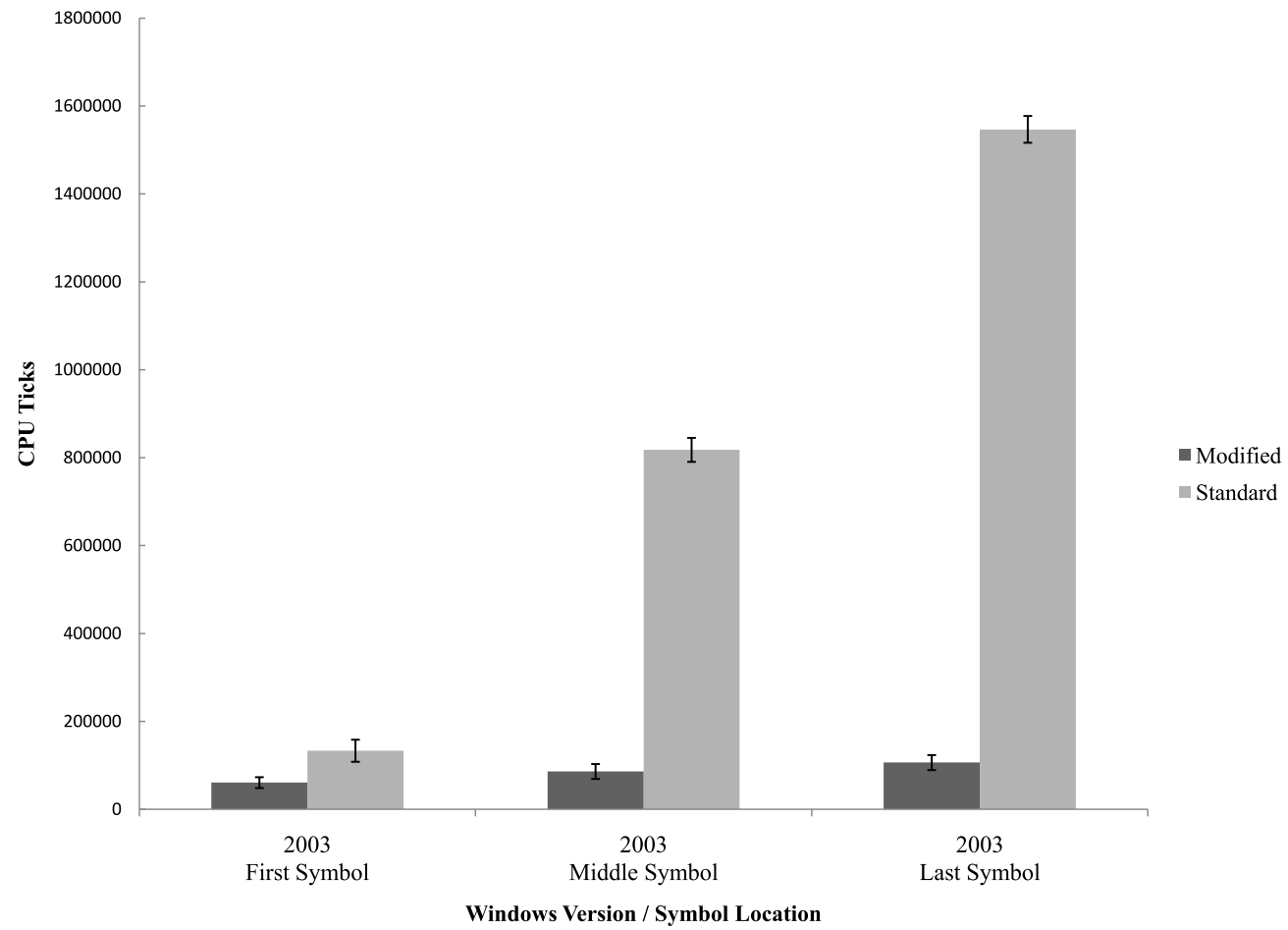


Figure A.7: Symbol Lookup Comparison for Windows 2003

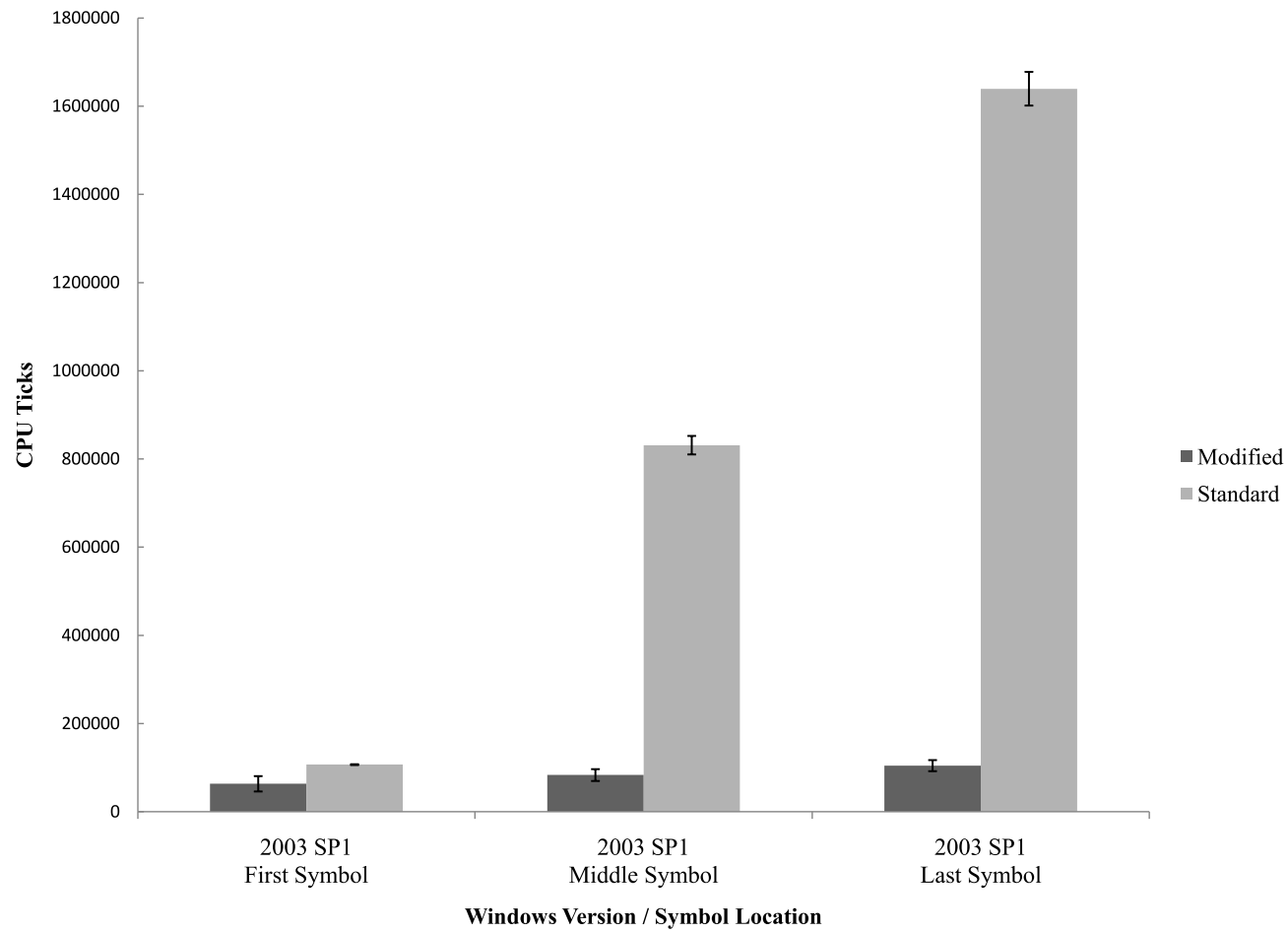


Figure A.8: Symbol Lookup Comparison for Windows 2003 SP1

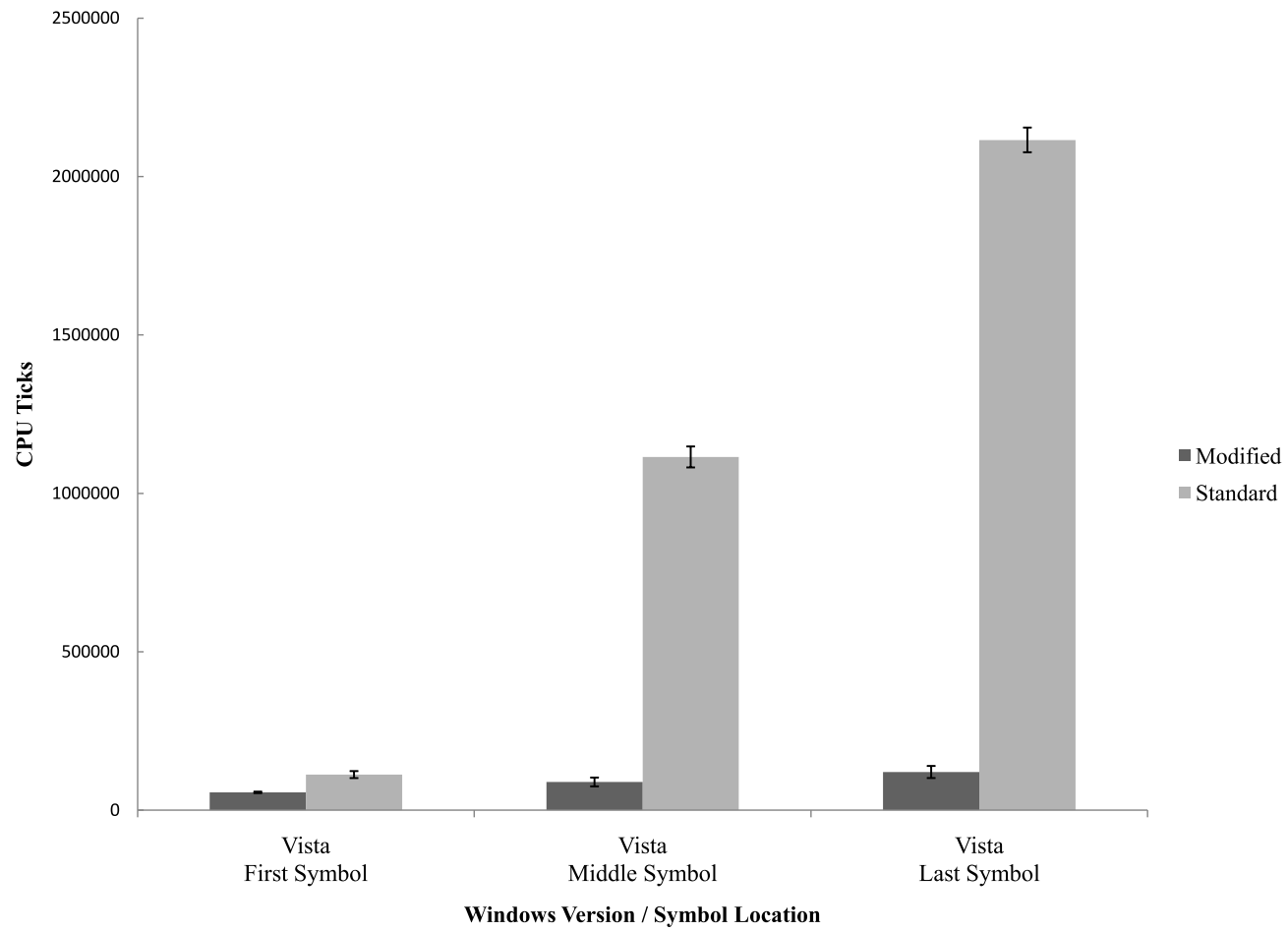


Figure A.9: Symbol Lookup Comparison for Windows Vista Business

Appendix B. Windows Operating System Data Structures

B.1 _EPROCESS Structure for Windows XP SP2

Listing B.1: Appendix2/xpsp2.txt

```
kd> dt -v -b _EPROCESS
ntdll!_EPROCESS
3 struct _EPROCESS, 107 elements, 0x260 bytes
    +0x000 Pcb : struct _KPROCESS, 29 elements, 0x6c bytes
    +0x000 Header : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
    +0x000 Type : UChar
    +0x001 Absolute : UChar
    8 +0x002 Size : UChar
    +0x003 Inserted : UChar
    +0x004 SignalState : Int4B
    +0x008 WaitListHead : struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x000 Flink : Ptr32 to
    13 +0x004 Blink : Ptr32 to
    +0x010 ProfileListHead : struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x000 Flink : Ptr32 to
    +0x004 Blink : Ptr32 to
    +0x018 DirectoryTableBase : (2 elements) UInt4B
    18 +0x020 LdtDescriptor : struct _KGDTENTRY, 3 elements, 0x8 bytes
    +0x000 LimitLow : UInt2B
    +0x002 BaseLow : UInt2B
    +0x004 HighWord : union __unnamed, 2 elements, 0x4 bytes
    +0x000 Bytes : struct __unnamed, 4 elements, 0x4 bytes
    23 +0x000 BaseMid : UChar
    +0x001 Flags1 : UChar
    +0x002 Flags2 : UChar
    +0x003 BaseHi : UChar
    +0x000 Bits : struct __unnamed, 10 elements, 0x4 bytes
    28 +0x000 BaseMid : Bitfield Pos 0, 8 Bits
    +0x000 Type : Bitfield Pos 8, 5 Bits
    +0x000 Dpl : Bitfield Pos 13, 2 Bits
    +0x000 Pres : Bitfield Pos 15, 1 Bit
    +0x000 LimitHi : Bitfield Pos 16, 4 Bits
    33 +0x000 Sys : Bitfield Pos 20, 1 Bit
    +0x000 Reserved_0 : Bitfield Pos 21, 1 Bit
    +0x000 Default_Big : Bitfield Pos 22, 1 Bit
    +0x000 Granularity : Bitfield Pos 23, 1 Bit
    +0x000 BaseHi : Bitfield Pos 24, 8 Bits
    38 +0x028 Int21Descriptor : struct _KIDTENTRY, 4 elements, 0x8 bytes
    +0x000 Offset : UInt2B
    +0x002 Selector : UInt2B
    +0x004 Access : UInt2B
    +0x006 ExtendedOffset : UInt2B
    43 +0x030 IopmOffset : UInt2B
    +0x032 Iopl : UChar
    +0x033 Unused : UChar
    +0x034 ActiveProcessors : UInt4B
    +0x038 KernelTime : UInt4B
    48 +0x03c UserTime : UInt4B
    +0x040 ReadyListHead : struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x000 Flink : Ptr32 to
    +0x004 Blink : Ptr32 to
    +0x048 SwapListEntry : struct _SINGLE_LIST_ENTRY, 1 elements, 0x4 bytes
    53 +0x000 Next : Ptr32 to
    +0x04c VdmTrampHandler : Ptr32 to
```

```

+0x050 ThreadListHead : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink           : Ptr32 to
+0x004 Blink           : Ptr32 to
58 +0x058 ProcessLock    : UInt4B
+0x05c Affinity        : UInt4B
+0x060 StackCount      : UInt2B
+0x062 BasePriority     : Char
+0x063 ThreadQuantum   : Char
63 +0x064 AutoAlignment  : UChar
+0x065 State           : UChar
+0x066 ThreadSeed      : UChar
+0x067 DisableBoost    : UChar
+0x068 PowerState      : UChar
68 +0x069 DisableQuantum : UChar
+0x06a IdealNode       : UChar
+0x06b Flags           : struct _KEXECUTE_OPTIONS, 7 elements, 0x1 bytes
+0x000 ExecuteDisable  : Bitfield Pos 0, 1 Bit
+0x000 ExecuteEnable   : Bitfield Pos 1, 1 Bit
73 +0x000 DisableThunkEmulation : Bitfield Pos 2, 1 Bit
+0x000 Permanent      : Bitfield Pos 3, 1 Bit
+0x000 ExecuteDispatchEnable : Bitfield Pos 4, 1 Bit
+0x000 ImageDispatchEnable : Bitfield Pos 5, 1 Bit
+0x000 Spare           : Bitfield Pos 6, 2 Bits
78 +0x06b ExecuteOptions : UChar
+0x06c ProcessLock     : struct _EX_PUSH_LOCK, 5 elements, 0x4 bytes
+0x000 Waiting         : Bitfield Pos 0, 1 Bit
+0x000 Exclusive       : Bitfield Pos 1, 1 Bit
+0x000 Shared          : Bitfield Pos 2, 30 Bits
83 +0x000 Value          : UInt4B
+0x000 Ptr             : Ptr32 to
+0x070 CreateTime      : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart         : UInt4B
+0x004 HighPart        : Int4B
88 +0x000 u              : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart         : UInt4B
+0x004 HighPart        : Int4B
+0x000 QuadPart        : Int8B
+0x078 ExitTime        : union _LARGE_INTEGER, 4 elements, 0x8 bytes
93 +0x000 LowPart         : UInt4B
+0x004 HighPart        : Int4B
+0x000 u              : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart         : UInt4B
+0x004 HighPart        : Int4B
98 +0x000 QuadPart        : Int8B
+0x080 RundownProtect  : struct _EX_RUNDOWN_REF, 2 elements, 0x4 bytes
+0x000 Count           : UInt4B
+0x000 Ptr             : Ptr32 to
+0x084 UniqueProcessId : Ptr32 to
103 +0x088 ActiveProcessLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink           : Ptr32 to
+0x004 Blink           : Ptr32 to
+0x090 QuotaUsage      : (3 elements) UInt4B
+0x09c QuotaPeak       : (3 elements) UInt4B
108 +0x0a8 CommitCharge    : UInt4B
+0x0ac PeakVirtualSize : UInt4B
+0x0b0 VirtualSize     : UInt4B
+0x0b4 SessionProcessLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink           : Ptr32 to
113 +0x004 Blink           : Ptr32 to
+0x0bc DebugPort       : Ptr32 to

```

```

+0x0c0 ExceptionPort      : Ptr32 to
+0x0c4 ObjectTable        : Ptr32 to
+0x0c8 Token              : struct _EX_FAST_REF, 3 elements, 0x4 bytes
118   +0x000 Object          : Ptr32 to
      +0x000 RefCnt        : Bitfield Pos 0, 3 Bits
      +0x000 Value         : Uint4B
+0x0cc WorkingSetLock     : struct _FAST_MUTEX, 5 elements, 0x20 bytes
      +0x000 Count         : Int4B
123   +0x004 Owner          : Ptr32 to
      +0x008 Contention    : Uint4B
      +0x00c Event         : struct _KEVENT, 1 elements, 0x10 bytes
      +0x000 Header        : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
128   +0x000 Type          : UChar
      +0x001 Absolute      : UChar
      +0x002 Size          : UChar
      +0x003 Inserted      : UChar
      +0x004 SignalState   : Int4B
      +0x008 WaitListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
133   +0x000 Flink         : Ptr32 to
      +0x004 Blink        : Ptr32 to
      +0x01c OldIrql       : Uint4B
+0x0ec WorkingSetPage     : Uint4B
+0x0f0 AddressCreationLock : struct _FAST_MUTEX, 5 elements, 0x20 bytes
138   +0x000 Count         : Int4B
      +0x004 Owner          : Ptr32 to
      +0x008 Contention    : Uint4B
      +0x00c Event         : struct _KEVENT, 1 elements, 0x10 bytes
      +0x000 Header        : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
143   +0x000 Type          : UChar
      +0x001 Absolute      : UChar
      +0x002 Size          : UChar
      +0x003 Inserted      : UChar
      +0x004 SignalState   : Int4B
148   +0x008 WaitListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
      +0x000 Flink         : Ptr32 to
      +0x004 Blink        : Ptr32 to
      +0x01c OldIrql       : Uint4B
+0x110 HyperSpaceLock     : Uint4B
153   +0x114 ForkInProgress : Ptr32 to
      +0x118 HardwareTrigger : Uint4B
      +0x11c VadRoot        : Ptr32 to
      +0x120 VadHint        : Ptr32 to
      +0x124 CloneRoot      : Ptr32 to
158   +0x128 NumberOfPrivatePages : Uint4B
      +0x12c NumberOfLockedPages : Uint4B
      +0x130 Win32Process    : Ptr32 to
      +0x134 Job             : Ptr32 to
      +0x138 SectionObject   : Ptr32 to
163   +0x13c SectionBaseAddress : Ptr32 to
      +0x140 QuotaBlock      : Ptr32 to
      +0x144 WorkingSetWatch : Ptr32 to
      +0x148 Win32WindowStation : Ptr32 to
      +0x14c InheritedFromUniqueProcessId : Ptr32 to
168   +0x150 LdtInformation   : Ptr32 to
      +0x154 VadFreeHint     : Ptr32 to
      +0x158 VdmObjects      : Ptr32 to
      +0x15c DeviceMap       : Ptr32 to
      +0x160 PhysicalVadList : struct _LIST_ENTRY, 2 elements, 0x8 bytes
173   +0x000 Flink         : Ptr32 to
      +0x004 Blink        : Ptr32 to

```

```

+0x168 PageDirectoryPte : struct _HARDWARE_PTE_X86, 13 elements, 0x4 bytes
+0x000 Valid           : Bitfield Pos 0, 1 Bit
+0x000 Write           : Bitfield Pos 1, 1 Bit
178 +0x000 Owner          : Bitfield Pos 2, 1 Bit
+0x000 WriteThrough    : Bitfield Pos 3, 1 Bit
+0x000 CacheDisable    : Bitfield Pos 4, 1 Bit
+0x000 Accessed        : Bitfield Pos 5, 1 Bit
+0x000 Dirty           : Bitfield Pos 6, 1 Bit
183 +0x000 LargePage      : Bitfield Pos 7, 1 Bit
+0x000 Global          : Bitfield Pos 8, 1 Bit
+0x000 CopyOnWrite     : Bitfield Pos 9, 1 Bit
+0x000 Prototype       : Bitfield Pos 10, 1 Bit
+0x000 reserved        : Bitfield Pos 11, 1 Bit
188 +0x000 PageFrameNumber : Bitfield Pos 12, 20 Bits
+0x168 Filler          : Uint8B
+0x170 Session         : Ptr32 to
+0x174 ImageFileName   : (16 elements) UChar
+0x184 JobLinks         : struct _LIST_ENTRY, 2 elements, 0x8 bytes
193 +0x000 Flink          : Ptr32 to
+0x004 Blink           : Ptr32 to
+0x18c LockedPagesList : Ptr32 to
+0x190 ThreadListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink           : Ptr32 to
198 +0x004 Blink          : Ptr32 to
+0x198 SecurityPort    : Ptr32 to
+0x19c PaeTop          : Ptr32 to
+0x1a0 ActiveThreads   : Uint4B
+0x1a4 GrantedAccess   : Uint4B
203 +0x1a8 DefaultHardErrorProcessing : Uint4B
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 PeB              : Ptr32 to
+0x1b4 PrefetchTrace   : struct _EX_FAST_REF, 3 elements, 0x4 bytes
+0x000 Object          : Ptr32 to
208 +0x000 RefCnt         : Bitfield Pos 0, 3 Bits
+0x000 Value           : Uint4B
+0x1b8 ReadOperationCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
213 +0x000 u              : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
+0x000 QuadPart        : Int8B
+0x1c0 WriteOperationCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
218 +0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
+0x000 u              : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
223 +0x000 QuadPart        : Int8B
+0x1c8 OtherOperationCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
+0x000 u              : struct __unnamed, 2 elements, 0x8 bytes
228 +0x000 LowPart         : Uint4B
+0x004 HighPart        : Int4B
+0x000 QuadPart        : Int8B
+0x1d0 ReadTransferCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart         : Uint4B
233 +0x004 HighPart        : Int4B
+0x000 u              : struct __unnamed, 2 elements, 0x8 bytes

```

```

+0x000 LowPart      : UInt4B
+0x004 HighPart     : Int4B
+0x000 QuadPart     : Int8B
238 +0x1d8 WriteTransferCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
+0x004 HighPart     : Int4B
+0x000 u            : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
243 +0x004 HighPart     : Int4B
+0x000 QuadPart     : Int8B
+0x1e0 OtherTransferCount : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
+0x004 HighPart     : Int4B
248 +0x000 u            : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
+0x004 HighPart     : Int4B
+0x000 QuadPart     : Int8B
+0x1e8 CommitChargeLimit : UInt4B
253 +0x1ec CommitChargePeak : UInt4B
+0x1f0 AweInfo       : Ptr32 to
+0x1f4 SeAuditProcessCreationInfo : struct _SE_AUDIT_PROCESS_CREATION_INFO, 1 elements, 0x4 bytes
+0x000 ImageFileName : Ptr32 to
+0x1f8 Vm            : struct _MMSUPPORT, 14 elements, 0x40 bytes
258 +0x000 LastTrimTime    : union _LARGE_INTEGER, 4 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
+0x004 HighPart     : Int4B
+0x000 u            : struct __unnamed, 2 elements, 0x8 bytes
+0x000 LowPart      : UInt4B
263 +0x004 HighPart     : Int4B
+0x000 QuadPart     : Int8B
+0x008 Flags        : struct _MMSUPPORT_FLAGS, 9 elements, 0x4 bytes
+0x000 SessionSpace : Bitfield Pos 0, 1 Bit
+0x000 BeingTrimmed : Bitfield Pos 1, 1 Bit
268 +0x000 SessionLeader  : Bitfield Pos 2, 1 Bit
+0x000 TrimHard     : Bitfield Pos 3, 1 Bit
+0x000 WorkingSetHard : Bitfield Pos 4, 1 Bit
+0x000 AddressSpaceBeingDeleted : Bitfield Pos 5, 1 Bit
+0x000 Available    : Bitfield Pos 6, 10 Bits
273 +0x000 AllowWorkingSetAdjustment : Bitfield Pos 16, 8 Bits
+0x000 MemoryPriority : Bitfield Pos 24, 8 Bits
+0x00c PageFaultCount : UInt4B
+0x010 PeakWorkingSetSize : UInt4B
+0x014 WorkingSetSize : UInt4B
278 +0x018 MinimumWorkingSetSize : UInt4B
+0x01c MaximumWorkingSetSize : UInt4B
+0x020 VmWorkingSetList : Ptr32 to
+0x024 WorkingSetExpansionLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink        : Ptr32 to
283 +0x004 Blink        : Ptr32 to
+0x02c Claim        : UInt4B
+0x030 NextEstimationSlot : UInt4B
+0x034 NextAgingSlot : UInt4B
+0x038 EstimatedAvailable : UInt4B
288 +0x03c GrowthSinceLastEstimate : UInt4B
+0x238 LastFaultCount : UInt4B
+0x23c ModifiedPageCount : UInt4B
+0x240 NumberOfVads : UInt4B
+0x244 JobStatus : UInt4B
293 +0x248 Flags : UInt4B
+0x248 CreateReported : Bitfield Pos 0, 1 Bit

```

```

+0x248 NoDebugInherit      : Bitfield Pos 1, 1 Bit
+0x248 ProcessExiting      : Bitfield Pos 2, 1 Bit
+0x248 ProcessDelete       : Bitfield Pos 3, 1 Bit
298 +0x248 Wow64SplitPages    : Bitfield Pos 4, 1 Bit
+0x248 VmDeleted           : Bitfield Pos 5, 1 Bit
+0x248 OutswapEnabled      : Bitfield Pos 6, 1 Bit
+0x248 Outswapped          : Bitfield Pos 7, 1 Bit
+0x248 ForkFailed          : Bitfield Pos 8, 1 Bit
303 +0x248 HasPhysicalVad     : Bitfield Pos 9, 1 Bit
+0x248 AddressSpaceInitialized : Bitfield Pos 10, 2 Bits
+0x248 SetTimerResolution  : Bitfield Pos 12, 1 Bit
+0x248 BreakOnTermination  : Bitfield Pos 13, 1 Bit
+0x248 SessionCreationUnderway : Bitfield Pos 14, 1 Bit
308 +0x248 WriteWatch        : Bitfield Pos 15, 1 Bit
+0x248 ProcessInSession    : Bitfield Pos 16, 1 Bit
+0x248 OverrideAddressSpace : Bitfield Pos 17, 1 Bit
+0x248 HasAddressSpace     : Bitfield Pos 18, 1 Bit
+0x248 LaunchPrefetched    : Bitfield Pos 19, 1 Bit
313 +0x248 InjectInpageErrors : Bitfield Pos 20, 1 Bit
+0x248 VmTopDown          : Bitfield Pos 21, 1 Bit
+0x248 Unused3            : Bitfield Pos 22, 1 Bit
+0x248 Unused4            : Bitfield Pos 23, 1 Bit
+0x248 VdmAllowed         : Bitfield Pos 24, 1 Bit
318 +0x248 Unused            : Bitfield Pos 25, 5 Bits
+0x248 Unused1            : Bitfield Pos 30, 1 Bit
+0x248 Unused2            : Bitfield Pos 31, 1 Bit
+0x24c ExitStatus          : Int4B
+0x250 NextPageColor       : UInt2B
323 +0x252 SubSystemMinorVersion : UChar
+0x253 SubSystemMajorVersion : UChar
+0x252 SubSystemVersion    : UInt2B
+0x254 PriorityClass       : UChar
+0x255 WorkingSetAcquiredUnsafe : UChar
328 +0x258 Cookie            : UInt4B
kd> dt -b -v _PEB
ntdll!_PEB
struct _PEB, 65 elements, 0x210 bytes
+0x000 InheritedAddressSpace : UChar
333 +0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged        : UChar
+0x003 SpareBool            : UChar
+0x004 Mutant                : Ptr32 to
+0x008 ImageBaseAddress     : Ptr32 to
338 +0x00c Ldr                  : Ptr32 to
+0x010 ProcessParameters    : Ptr32 to
+0x014 SubSystemData        : Ptr32 to
+0x018 ProcessHeap          : Ptr32 to
+0x01c FastPebLock          : Ptr32 to
343 +0x020 FastPebLockRoutine   : Ptr32 to
+0x024 FastPebUnlockRoutine : Ptr32 to
+0x028 EnvironmentUpdateCount : UInt4B
+0x02c KernelCallbackTable  : Ptr32 to
+0x030 SystemReserved       : (1 elements) UInt4B
348 +0x034 AtlThunkSListPtr32   : UInt4B
+0x038 FreeList             : Ptr32 to
+0x03c TlsExpansionCounter  : UInt4B
+0x040 TlsBitmap            : Ptr32 to

```

B.2 *_KPCR Structure*

Listing B.2: Appendix2/kpcr.txt

```
kd> dt _KPCR
nt!_KPCR
+0x000 NtTib : _NT_TIB
4 +0x01c SelfPcr : Ptr32 _KPCR
+0x020 Prcb : Ptr32 _KPRCB
+0x024 Irql : UChar
+0x028 IRR : Uint4B
+0x02c IrrActive : Uint4B
9 +0x030 IDR : Uint4B
+0x034 KdVersionBlock : Ptr32 Void
+0x038 IDT : Ptr32 _KIDTENTRY
+0x03c GDT : Ptr32 _KGDENTRY
+0x040 TSS : Ptr32 _KTSS
14 +0x044 MajorVersion : Uint2B
+0x046 MinorVersion : Uint2B
+0x048 SetMember : Uint4B
+0x04c StallScaleFactor : Uint4B
+0x050 DebugActive : UChar
19 +0x051 Number : UChar
+0x052 Spare0 : UChar
+0x053 SecondLevelCacheAssociativity : UChar
+0x054 VdmAlert : Uint4B
+0x058 KernelReserved : [14] Uint4B
24 +0x090 SecondLevelCacheSize : Uint4B
+0x094 HalReserved : [16] Uint4B
+0x0d4 InterruptMode : Uint4B
+0x0d8 Spare1 : UChar
+0x0dc KernelReserved2 : [17] Uint4B
29 +0x120 PrcbData : _KPRCB
```

B.3 *_DBGKD_GET_VERSION64 Structure*

Listing B.3: Appendix2/dbgkd.txt

```
1 kd> dt _DBGKD_GET_VERSION64
nt!_DBGKD_GET_VERSION64
+0x000 MajorVersion : Uint2B
+0x002 MinorVersion : Uint2B
+0x004 ProtocolVersion : Uint2B
6 +0x006 Flags : Uint2B
+0x008 MachineType : Uint2B
+0x00a MaxPacketType : UChar
+0x00b MaxStateChange : UChar
+0x00c MaxManipulate : UChar
11 +0x00d Simulation : UChar
+0x00e Unused : [1] Uint2B
+0x010 KernBase : Uint8B
+0x018 PsLoadedModuleList : Uint8B
+0x020 DebuggerDataList : Uint8B
```

B.4 *_KDDEBUGGER_DATA64* Structure

Listing B.4: Appendix2/kddebuggerdata64.txt

```
typedef struct _KDDEBUGGER_DATA64
{
    DBGKD_DEBUG_DATA_HEADER64 Header;
    ULONG64 KernBase;
5    GCC_ULONG64 BreakpointWithStatus;
    ULONG64 SavedContext;
    USHORT ThCallbackStack;
    USHORT NextCallback;
    USHORT FramePointer;
10    USHORT PaeEnabled:1;
    GCC_ULONG64 KiCallUserMode;
    GCC_ULONG64 KeUserCallbackDispatcher;
    GCC_ULONG64 PsLoadedModuleList;
    GCC_ULONG64 PsActiveProcessHead;
15    GCC_ULONG64 PspCidTable;
    GCC_ULONG64 ExpSystemResourcesList;
    GCC_ULONG64 ExpPagedPoolDescriptor;
    GCC_ULONG64 ExpNumberOfPagedPools;
    GCC_ULONG64 KeTimeIncrement;
20    GCC_ULONG64 KeBugCheckCallbackListHead;
    GCC_ULONG64 KiBugcheckData;
    GCC_ULONG64 IopErrorLogListHead;
    GCC_ULONG64 ObpRootDirectoryObject;
    GCC_ULONG64 ObpTypeObjectType;
25    GCC_ULONG64 MmSystemCacheStart;
    GCC_ULONG64 MmSystemCacheEnd;
    GCC_ULONG64 MmSystemCacheWs;
    GCC_ULONG64 MmPfnDatabase;
    GCC_ULONG64 MmSystemPtesStart;
30    GCC_ULONG64 MmSystemPtesEnd;
    GCC_ULONG64 MmSubsectionBase;
    GCC_ULONG64 MmNumberOfPagingFiles;
    GCC_ULONG64 MmLowestPhysicalPage;
    GCC_ULONG64 MmHighestPhysicalPage;
35    GCC_ULONG64 MmNumberOfPhysicalPages;
    GCC_ULONG64 MmMaximumNonPagedPoolInBytes;
    GCC_ULONG64 MmNonPagedSystemStart;
    GCC_ULONG64 MmNonPagedPoolStart;
    GCC_ULONG64 MmNonPagedPoolEnd;
40    GCC_ULONG64 MmPagedPoolStart;
    GCC_ULONG64 MmPagedPoolEnd;
    GCC_ULONG64 MmPagedPoolInformation;
    ULONG64 MmPageSize;
    GCC_ULONG64 MmSizeOfPagedPoolInBytes;
45    GCC_ULONG64 MmTotalCommitLimit;
    GCC_ULONG64 MmTotalCommittedPages;
    GCC_ULONG64 MmSharedCommit;
    GCC_ULONG64 MmDriverCommit;
    GCC_ULONG64 MmProcessCommit;
50    GCC_ULONG64 MmPagedPoolCommit;
    GCC_ULONG64 MmExtendedCommit;
    GCC_ULONG64 MmZeroedPageListHead;
    GCC_ULONG64 MmFreePageListHead;
    GCC_ULONG64 MmStandbyPageListHead;
55    GCC_ULONG64 MmModifiedPageListHead;
```

```

GCC_ULONG64 MmModifiedNoWritePageListHead;
GCC_ULONG64 MmAvailablePages;
GCC_ULONG64 MmResidentAvailablePages;
GCC_ULONG64 PoolTrackTable;
60 GCC_ULONG64 NonPagedPoolDescriptor;
GCC_ULONG64 MmHighestUserAddress;
GCC_ULONG64 MmSystemRangeStart;
GCC_ULONG64 MmUserProbeAddress;
GCC_ULONG64 KdPrintCircularBuffer;
65 GCC_ULONG64 KdPrintCircularBufferEnd;
GCC_ULONG64 KdPrintWritePointer;
GCC_ULONG64 KdPrintRolloverCount;
GCC_ULONG64 MmLoadedUserImageList;
GCC_ULONG64 NtBuildLab;
70 GCC_ULONG64 KiNormalSystemCall;
GCC_ULONG64 KiProcessorBlock;
GCC_ULONG64 MmUnloadedDrivers;
GCC_ULONG64 MmLastUnloadedDriver;
GCC_ULONG64 MmTriageActionTaken;
75 GCC_ULONG64 MmSpecialPoolTag;
GCC_ULONG64 KernelVerifier;
GCC_ULONG64 MmVerifierData;
GCC_ULONG64 MmAllocatedNonPagedPool;
GCC_ULONG64 MmPeakCommitment;
80 GCC_ULONG64 MmTotalCommitLimitMaximum;
GCC_ULONG64 CmNtCSDVersion;
GCC_ULONG64 MmPhysicalMemoryBlock;
GCC_ULONG64 MmSessionBase;
GCC_ULONG64 MmSessionSize;
85 GCC_ULONG64 MmSystemParentTablePage;
GCC_ULONG64 MmVirtualTranslationBase;
USHORT OffsetKThreadNextProcessor;
USHORT OffsetKThreadTeb;
USHORT OffsetKThreadKernelStack;
90 USHORT OffsetKThreadInitialStack;
USHORT OffsetKThreadApcProcess;
USHORT OffsetKThreadState;
USHORT OffsetKThreadBStore;
USHORT OffsetKThreadBStoreLimit;
95 USHORT SizeEProcess;
USHORT OffsetEprocessPeb;
USHORT OffsetEprocessParentCID;
USHORT OffsetEprocessDirectoryTableBase;
USHORT SizePrCb;
100 USHORT OffsetPrCbDpcRoutine;
USHORT OffsetPrCbCurrentThread;
USHORT OffsetPrCbMhz;
USHORT OffsetPrCbCpuType;
USHORT OffsetPrCbVendorString;
105 USHORT OffsetPrCbProcStateContext;
USHORT OffsetPrCbNumber;
USHORT SizeEThread;
GCC_ULONG64 KdPrintCircularBufferPtr;
GCC_ULONG64 KdPrintBufferSize;
110 GCC_ULONG64 KeLoaderBlock;
USHORT SizePcr;
USHORT OffsetPcrSelfPcr;
USHORT OffsetPcrCurrentPrCb;
USHORT OffsetPcrContainedPrCb;
115 USHORT OffsetPcrInitialBStore;

```

```

    USHORT OffsetPcrBStoreLimit;
    USHORT OffsetPcrInitialStack;
    USHORT OffsetPcrStackLimit;
    USHORT OffsetPrCbPcrPage;
120    USHORT OffsetPrCbProcStateSpecialReg;
    USHORT GdtR0Code;
    USHORT GdtR0Data;
    USHORT GdtR0Pcr;
    USHORT GdtR3Code;
125    USHORT GdtR3Data;
    USHORT GdtR3Teb;
    USHORT GdtLdt;
    USHORT GdtTss;
    USHORT Gdt64R3CmCode;
130    USHORT Gdt64R3CmTeb;
    GCC_ULONGLONG IopNumTriageDumpDataBlocks;
    GCC_ULONGLONG IopTriageDumpDataBlocks;
    GCC_ULONGLONG VfCrashDataBlock;
} KDDEBUGGER_DATA64, *PKDDEBUGGER_DATA64;

```

Appendix C. Building the Test Platform

This appendix gives a detailed description of how to build and configure the Xen environment as described in this thesis. A Dell Latitude D630 with an Intel Core 2 Duo T7300 and 2 GB of memory is used to carry out experimentation.

C.1 Install Fedora 8

1. Install Fedora 8 with the software development tools option. Also, during installation disable 'IPv6', the built-in firewall and SELinux.
2. Add `/sbin` and `/usr/sbin` to the path statement in `.bash_profile` for root and the configured user account.
3. Install dev86 tools from a root terminal with: `yum install dev86`
4. Install the network bridge utilities necessary for Xen with:
`yum install bridge-utils`
5. Disable tls with: `mv /lib/tls /lib/tls.disabled`
6. Remove nag about tls during boot with:
`echo "hwcap 0 nosegneq" >> /etc/ld.so.conf`
7. Update the dynamic linker with: `ldconfig`

C.2 Install Xen 3.1.4

1. Download the Xen 3.1.4 source code from the www.xen.org archives at <http://bits.xensource.com/oss-xen/release/3.1.4/xen-3.1.4.tar.gz>
2. Unpack the gzipped tarball with: `tar xzf xen-3.1.4.tar.gz`
3. Change to the xen directory with: `cd xen-3.1.4`
4. Compile Xen with: `make world`
This will take some time and you must be connected to the Internet because the Linux 2.6.18.8 kernel is downloaded.
5. Install Xen with: `make install`

6. Execute the install script with: `./install.sh`

C.3 Configuring Xen Boot

1. Create the dependency file for the 2.6.18.8-xen kernel with:

```
/sbin/depmod 2.6.18.8-xen
```

2. Create the initrd image for 2.6.18.8-xen with:

```
/sbin/mkinitrd -v -f --with=aacraid --with=sd_mod --with=scsi_mod  
--with=tun /boot/initrd-2.6.18.8-xen.img 2.6.18.8-xen
```

3. To the file `/boot/grub/menu.lst` add the following:

```
title Xen 3.1.4 / XenLinux 2.6.18.8  
root(hd0,1)  
kernel /xen-3.1.gz console=vga dom0_mem=512M  
module /vmlinuz-2.6.18.8-xen root=/dev/VolGroup00/LogVol100 \  
ro console=tty0  
module /initrd-2.6.18.8-xen.img
```

The boot menu will now contain the option to boot the Xen kernel. The Xen kernel does not support the video card in the Dell D630 so upon initial boot the graphical user interface will fail to come up. The operating system is able to configure the generic VESA driver by choosing the default options in the recovery dialog. The changes are made permanently in the `/etc/X11/xorg.conf` file and the problem should not reappear.

C.4 *XenAccess*

XenAccess can be obtained from its Google code repository at <http://xenaccess.googlecode.com/svn/trunk>. However, this version does not incorporate my modifications which were made to revision 130. This version can be checked-out from the Fedora 8 command line using the command:

```
svn checkout -r 130 <XenAccess URL> <destination folder>
```

Requests for the modified source code used in this research should be directed to Dr. Barry Mullins, barry.mullins@afit.edu.

1. Unpack the gzipped tarball with: `tar xzf filename.tar.gz`
2. Change to the `libxa` directory.
3. Compile the library with: `make`
4. Install the library with: `make install`

In the source code for this research modifications were made to many library files, but the `page1.c` file in the `xenaccess` directory contains the bulk of the additions made by the author. Additionally, the `examples` folder contains the test programs.

C.5 *Creating a HVM DomU for Xen*

This section describes in detail how to create a virtual machine running HVM for use with Xen. Creation of the configuration file and virtual disk files is the same regardless of the guest OS to be installed.

1. Creating a virtual disk file for the VM named `imagename.img` and 10GB in size:

```
dd if=/dev/zero of=imagename.img bs=1k seek=10240k count=1
```
2. An example Xen HVM configuration file can be found in `/etc/xen/` after installation

Bibliography

- AA06. Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.
- ADC05. Tim Abels, Puneet Dhawan, and Balasubramanian Chandrasekaran. An Overview of Xen Virtualization, 2005. Available from: <http://www.dell.com/downloads/global/power/ps3q05-20050191-Abels.pdf>.
- AMD08. Inc. Advanced Micro Devices. AMD64 Architecture Programmer's Manual, 2008. Available from: <http://developer.intel.com/products/processor/manuals/index.htm>.
- App08. Net Applications. Operating system market share, 2008. Available from: <http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8>.
- Bar. Edgar Barbosa. Finding some non-exported kernel variables in windows xp. Available from: <http://www.rootkit.com/vault/Opc0de/GetVarXP.pdf>.
- BDF⁺03. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.
- C01. C0ldCrow. Windows kernelmode, keyboard independent keylogger. Available from: <http://www.awarenetwork.org/etc/beta/?x=1>.
- Car06. Harlan Carvey. Windows incident response blog. os detection, explained, 2006. Available from: <http://windowsir.blogspot.com/2006/09/os-detection-explained.html>.
- Car07. Harlan Carvey. *Windows Forensic Analysis*. Syngress, 2007.
- CC03. Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.
- Chu06. Simon P. Chung. On the (im)practicality of system-call-based ids, 2006. Available from: <http://www.cs.utexas.edu/users/phchung/strike.ppt>.
- CN01. P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138, May 2001.

- Cor08a. Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2008. Available from: <http://developer.intel.com/products/processor/manuals/index.htm>.
- Cor08b. Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2008. Available from: <http://developer.intel.com/products/processor/manuals/index.htm>.
- Cor08c. Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2008. Available from: <http://developer.intel.com/products/processor/manuals/index.htm>.
- Cor08d. Microsoft Corporation. Microsoft portable executable and common object file format specification, 2008. Available from: <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>.
- Cor08e. Microsoft Corporation. VS_VERSIONINFO Structure, 2008. Available from: <http://msdn.microsoft.com/en-us/library/ms647001.aspx>.
- CS09. Inc. Citrix Systems. Xen overview, 2009. Available from: <http://www.xen.org/about/>.
- DPB99. Prasad Dabak, Sandeep Phadke, and Milind Borate. *Undocumented Windows NT*. John Wiley & Sons, 1999.
- FKF⁺03. H.H. Feng, O.M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong. Anomaly detection using call stack information. *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 62–75, May 2003.
- FS08. Christof Fetzer and Martin Süßkraut. Switchblade: enforcing dynamic personalized system call models. *SIGOPS Oper. Syst. Rev.*, 42(4):273–286, 2008.
- Gar03. Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.
- GAWF07. Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.
- GR03. Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.
- HB05. Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
- HFS98. Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

- Int06. Intox. Agony Ring0 Rootkit, 2006. Available from:
<http://www.opensc.ws/c-c/1682-agony-ring0-rootkit.html>.
- Ion. Alex Ionescu. Getting kernel variables from kdversionblock, part 2. Available from: <http://www.rootkit.com/newsread.php?newsid=153>.
- JADAD06. Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX 2006 Annual Technical Conference (USENIX '06)*, Boston, MA, June 2006.
- JADAD08. Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM.
- JFMA04. Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, San Diego, CA, August 2004.
- Jiu08. Jiurl. JiurlPortHide Source Code. Programmers United Develop Net., 2008. Available from: <http://www.pudn.com/downloads114/sourcecode/windows/system/detail481865.html>.
- Jon07. Stephen T. Jones. *Implicit Operating System Awareness in a Virtual Machine Monitor*. PhD thesis, University of Wisconsin – Madison, April 2007.
- JWX07. Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based ”out-of-the-box” semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM.
- KS07. Aditya Kapoor and Ahmed Sallam. White paper: Rootkits part 2: A technical primer, 2007. Available from: http://www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf.
- Lin. Mission Critical Linux. Crash core analysis suite. Available from: <http://oss.missioncriticallinux.com/projects/crash/>.
- McGa. Roland McGrath. strace. Available from: <http://sourceforge.net/projects/strace/>.
- McGb. Roland McGrath. utrace. Available from: <http://people.redhat.com/roland/utrace/>.
- Mot07. Stephen Mott. Exploring hardware-based primitives to enhance parallel security monitoring in a novel computing architecture. Master’s thesis,

Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, March 2007.

- MY07. Michael Myers and Stephen Youndt. An introduction to hardware-assisted virtual machine (hvm) rootkits, 2007. Available from: <http://www.crucialsecurity.com/documents/hvmrootkits.pdf>.
- Pag08. Brett Pagel. Email communication with bryan payne, 2008.
- Pay07. Bryan Payne. Xenaccess library, 2007. Available from: <http://www.xenaccess.org>.
- PCL07. Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 385–397, December 2007.
- PCSL08. Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 233–247, May 2008.
- PFH⁺05. I. Pratt, Fraser, Hand, Limpach, Warfield, Magenheimer, Nakajima, and Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of the Linux Symposium: Volume 2*, pages 65–78, Ottawa, Ontario, Canada, 2005. Linux Symposium, Inc.
- Pro03. N. Provos. Improving host security with system call policies. *12th USENIX Security Symposium*, pages 257–271, 2003 2003. PT: C; CT: 12th USENIX Security Symposium; CY: 4-8 Aug. 2003; CL: Washington, DC, USA; SP: USENIX; PV: Berkeley, CA, USA; NR: 39.
- Pro09. The Metasploit Project. Windows system call table, 2009. Available from: <http://www.metasploit.com/users/opcode/syscalls.html>.
- PSJ08. Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 156–167, New York, NY, USA, 2008. ACM.
- Rie06. Chris Ries. Inside windows rootkits, 2006. Available from: http://www.vigilantminds.com/files/inside_windows_rootkits.pdf.
- RS04. Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

- Rut06. Joanna Rutkowska. Introducing stealth malware taxonomy, 2006. Available from: <http://invisiblethings.org/papers/malware-taxonomy.pdf>.
- Rut07a. Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition tools (part i: Amd case), 2007. Available from: <http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt>.
- Rut07b. Joanna Rutkowska. Virtualization - the other side of the coin, 2007. Available from: <http://www.invisiblethings.org/papers/NLUUG-virtualization.ppt>.
- Rut08. Joanna Rutkowska. Security Challenges in Virtualized Environments, 2008. Available from: <http://invisiblethings.org/papers.html>.
- SBDB01. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 144–155, 2001.
- Sec08. Packet Storm Security. Packet storm security exploit database, 2008. Available from: <http://packetstormsecurity.nl/assess/exploits/>.
- Sou. Inc. Sourcefire. Snort. Available from: <http://www.snort.org>.
- SYfZ+05. Yue Shen, Fei Yu, Ling fen Zhang, Ji yao An, and Miao liang Zhu. An intrusion detection system based on system call. *Internet, 2005. The First IEEE and IFIP International Conference in Central Asia on*, pages 159–184, 26-29 Sept. 2005.
- Sym08. Symantec. Symantec virus definitions, 2008. Available from: http://www.symantec.com/business/security_response/definitions.jsp.
- SZ03. Ed Skoudis and Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- UNR+05. R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.
- vH08. William von Hagen. *Professional Xen Virtualization*. Wrox Press, 2008.
- Wat07. Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.
- WD01. D. Wagner and R. Dean. Intrusion detection via static analysis. *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168, 2001.

- Wil05. Paul D. Williams. *CuPIDS: Increasing Information System Security through the Use of Dedicated Co-processing*. PhD thesis, Purdue, July 2005.
- WS02. David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.
- YA04. M.M. Yasin and A.A. Awan. A study of host-based ids using system calls. *Networking and Communication, 2004. INCC 204. International Conference on*, pages 36–41, 11-13 June 2004.
- YXS⁺05. Fei Yu, Cheng Xu, Yue Shen, Ji yao An, and Lin feng Zhang. Intrusion detection based on system call finite-state automation machine. *Industrial Technology, 2005. ICIT 2005. IEEE International Conference on*, pages 63–68, 14-17 Dec. 2005.
- Zov06. Dino A. Dai Zovi. Hardware virtualization rootkits, 2006. Available from: http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf.

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY)		2. REPORT TYPE		3. DATES COVERED (From — To)		
24-02-2009		Master's Thesis		Jul 2007 — Mar 2009		
4. TITLE AND SUBTITLE				5a. CONTRACT NUMBER		
Automated Virtual Machine Introspection for Host-Based Intrusion Detection				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S)				5d. PROJECT NUMBER		
Brett A. Pagel, Capt, USAF				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)					8. PERFORMING ORGANIZATION REPORT NUMBER	
Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765					AFIT/GCE/ENG/09-07	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)					10. SPONSOR/MONITOR'S ACRONYM(S)	
N/A					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT						
Approval for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT						
<p>This thesis examines techniques to automate configuration of an intrusion detection system utilizing hardware-assisted virtualization. These techniques are used to detect the version of a running guest operating system, automatically configure version-specific operating system information needed by the introspection library, and to locate and monitor important operating system data structures. This research simplifies introspection library configuration and is a step toward operating system independent introspection. An operating system detection algorithm and Windows virtual machine system service dispatch table monitor are implemented using the Xen hypervisor and a modified version of the XenAccess library. All detection and monitoring is implemented from the Xen management domain. Results of the operating system detection are used to initialize the XenAccess library. Library initialization time and kernel symbol retrieval are compared to the standard library. The algorithm is evaluated using nine versions of the Windows operating system. The system service dispatch table monitor is evaluated using the Agony and ProAgent rootkits. The automation techniques successfully detect the operating system and system service dispatch table hooks for the nine Windows versions tested. The modified XenAccess library exhibits an average initialization speedup of 1.9. Kernel symbol lookup is 10 times faster, on average. The hook detector is able to detect all hooks used by both rootkits.</p>						
15. SUBJECT TERMS						
operating systems, intrusion detection, computer viruses						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT		18. NUMBER OF PAGES	
a. REPORT	b. ABSTRACT	c. THIS PAGE			19a. NAME OF RESPONSIBLE PERSON	
U	U	U	UU		Dr. Barry Mullins	
					19b. TELEPHONE NUMBER (include area code)	
					(937) 255-3636, ext 7979; barry.mullins@afit.edu	